©1996 por The Massachusetts Institute of Technology

Estructura e Interpretación de Programas de Computadoras,
segunda edición

Harold Abelson y Gerald Jay Sussman
con Julie Sussman, prólogo de Alan J. Perlis

Este trabajo está licenciado bajo una Creative Commons
Licencia Internacional Atribución-CompartirIgual 4.0
(CC BY-SA 4.0). Basado en un trabajo en mitpress.mit.edu.

The MIT Press
Cambridge, Massachusetts
Londres, Inglaterra

McGraw-Hill Book Company
Nueva York, St. Louis, San Francisco,
Montreal, Toronto

Formato Texinfo no oficial 2.andresraba5.6 (Febrero 2, 2016),
basado en 2.neilvandyke4 (10 de enero de 2007).

Formato Texinfo no oficial

Este es el libro de la segunda edición de SICP, del formato Texinfo no oficial.

Probablemente lo esté leyendo en un navegador de hipertexto Info, como el modo Info de Emacs. Alternativamente, podría estar leyéndolo formateado en TeX en su pantalla o impresora, aunque eso sería una tontería. Y, si se imprime, costoso.

El formato oficial de HTML-y-GIF, distribuido libremente, fue convertido personalmente por primera vez al formato Texinfo no oficial (UTF) versión 1 por Lytha Ayth durante un largo fin de semana de amorío con Emacs en abril de 2001.

El UTF es más fácil de buscar que el formato HTML. También es mucho más accesible para personas que usan ordenadores modestos, como PCs basados en ’386 donados. Un 386 puede, en teoría, ejecutar Linux, Emacs y un intérprete de Scheme simultáneamente, pero la mayoría de los 386 probablemente no puedan ejecutar también Netscape y el necesario X Window System sin introducir prematuramente a los jóvenes hackers con fondos limitados al concepto de *thrashing*. El UTF también puede caber sin comprimir en un disquete de 1.44MB, lo que puede ser útil para instalar UTF en PCs que no tienen acceso a Internet o a una LAN.

La conversión a Texinfo ha sido una transliteración directa, en la medida de lo posible. Al igual que la conversión de TeX-a-HTML, esto no estuvo exento de la introducción de algunas roturas. En el caso del formato Texinfo no oficial, las figuras han sufrido una resurrección amateur del arte perdido del ASCII. Además, es bastante posible que se introdujeran algunos errores de ambigüedad durante la conversión de algunos de los copiosos superíndices (‘^’) y subíndices (‘_’). Discernir *cuáles* se ha dejado como ejercicio para el lector. Pero al menos no ponemos en riesgo a nuestros valientes astronautas codificando el símbolo de *mayor o igual que* como `<u>></u>`.

Si modifica `sicp.texi` para corregir errores o mejorar el arte ASCII, entonces actualice la línea `@set utfversion 2.andresraba5.6` para reflejar su delta.

Por ejemplo, si comenzó con la versión `1` de Lytha, y su nombre es Bob, entonces podría nombrar sus sucesivas versiones `1.bob1`, `1.bob2`, ... `1.bobn`. También actualice `utfversiondate`. Si desea distribuir su versión en la Web, entonces incrustar la cadena "sicp.texi" en algún lugar del archivo o de la página web facilitará que la gente lo encuentre con los motores de búsqueda web.

Se cree que el formato Texinfo no oficial está en consonancia con el espíritu de la versión HTML distribuida gratuitamente. Pero nunca se sabe cuándo la armada de abogados de alguien podría necesitar algo que hacer, y ponerse quisquillosos por alguna pequeña cosa benigna, así que piénselo dos veces antes de usar su nombre completo o distribuir formatos Info, DVI, PostScript o PDF que puedan incrustar su cuenta o el nombre de su máquina.

Peath, Lytha Ayth

Addendum: Vea también las videoconferencias de SICP de Abelson y Sussman:
en MIT CSAIL o MIT OCW.

Segundo Addendum: Arriba está la introducción original al UTF de 2001. Diez años después, el UTF ha sido transformado: los símbolos y fórmulas matemáticas están correctamente compuestos, y las figuras dibujadas en gráficos vectoriales. Las fórmulas de texto originales y las figuras de arte ASCII todavía están allí en la fuente Texinfo, pero solo se mostrarán cuando se compilen a salida Info. En los albores de los lectores de libros electrónicos y las tabletas, leer un PDF en pantalla oficialmente ya no es una tontería. ¡Disfruten!

A.R, mayo de 2011

Dedicatoria

ESTE LIBRO ESTÁ DEDICADO, con respeto y admiración, al espíritu que vive en el ordenador.

> “Creo que es extraordinariamente importante que nosotros, en la informática, mantengamos la diversión en la computación. Cuando comenzó, era muy divertido. Por supuesto, los clientes de pago eran estafados de vez en cuando, y después de un tiempo empezamos a tomar sus quejas en serio. Empezamos a sentir como si realmente fuéramos responsables del uso exitoso, sin errores y perfecto de estas máquinas. No creo que lo seamos. Creo que somos responsables de expandirlas, de ponerlas en nuevas direcciones y de mantener la diversión en casa. Espero que el campo de la informática nunca pierda su sentido del humor. Sobre todo, espero que no nos convirtamos en misioneros. No se sientan como si fueran vendedores de Biblias. El mundo ya tiene demasiados de esos. Lo que saben sobre computación, otras personas lo aprenderán. No sientan como si la clave para una computación exitosa estuviera solo en sus manos. Lo que está en sus manos, creo y espero, es la inteligencia: la habilidad de ver la máquina como algo más de lo que era cuando se la presentaron por primera vez, que pueden hacerla más.”
>
> —Alan J. Perlis (1 de abril de 1922 – 7 de febrero de 1990)

Prólogo

EDUCADORES, GENERALES, DIETISTAS, psicólogos y padres programan. Ejércitos, estudiantes y algunas sociedades son programados. Un asalto a grandes problemas emplea una sucesión de programas, la mayoría de los cuales surgen en el camino. Estos programas están plagados de problemas que parecen ser particulares del problema en cuestión. Para apreciar la programación como una actividad intelectual por derecho propio, debe recurrir a la programación de ordenadores; debe leer y escribir programas de ordenador, muchos de ellos. No importa mucho de qué tratan los programas o a qué aplicaciones sirven. Lo que importa es qué tan bien funcionan y qué tan fluidamente se integran con otros programas en la creación de programas aún mayores. El programador debe buscar tanto la perfección de la parte como la adecuación del conjunto. En este libro, el uso de "programa" se centra en la creación, ejecución y estudio de programas escritos en un dialecto de Lisp para su ejecución en un ordenador digital. Al usar Lisp, no restringimos ni limitamos lo que podemos programar, sino solo la notación para las descripciones de nuestros programas.

Nuestro trato con la materia de este libro nos involucra con tres focos de fenómenos: la mente humana, colecciones de programas de ordenador y el ordenador. Cada programa de ordenador es un modelo, concebido en la mente, de un proceso real o mental. Estos procesos, que surgen de la experiencia y el pensamiento humanos, son enormes en número, intrincados en detalle y, en cualquier momento, solo se comprenden parcialmente. Rara vez son modelados a nuestra permanente satisfacción por nuestros programas de ordenador. Así, aunque nuestros programas son colecciones discretas de símbolos cuidadosamente elaboradas, mosaicos de funciones entrelazadas, evolucionan continuamente: los cambiamos a medida que nuestra percepción del modelo se profundiza, se amplía, se generaliza hasta que el modelo alcanza finalmente un lugar metaestable dentro de otro modelo con el que luchamos. La fuente de la euforia asociada con la programación de ordenadores es el continuo despliegue en la mente y en el ordenador de mecanismos expresados como programas y la explosión de percepción que generan. Si el arte interpreta nuestros sueños, ¡el ordenador los ejecuta bajo

la apariencia de programas!

A pesar de todo su poder, el ordenador es un capataz severo. Sus programas deben ser correctos, y lo que deseamos decir debe decirse con precisión en cada detalle. Como en cualquier otra actividad simbólica, nos convencemos de la verdad de un programa a través del argumento. A Lisp mismo se le puede asignar una semántica (otro modelo, por cierto), y si la función de un programa puede especificarse, digamos, en el cálculo de predicados, los métodos de prueba de la lógica pueden usarse para hacer un argumento de corrección aceptable. Desafortunadamente, a medida que los programas se vuelven grandes y complicados, como casi siempre lo hacen, la adecuación, consistencia y corrección de las especificaciones mismas se vuelven dudosas, de modo que los argumentos formales completos de corrección rara vez acompañan a los grandes programas. Dado que los programas grandes crecen a partir de los pequeños, es crucial que desarrollemos un arsenal de estructuras de programas estándar de cuya corrección nos hayamos asegurado —las llamamos modismos— y aprendamos a combinarlas en estructuras más grandes utilizando técnicas de organización de valor probado. Estas técnicas se tratan extensamente en este libro, y comprenderlas es esencial para participar en la empresa prometeica llamada programación. Más que cualquier otra cosa, el descubrimiento y el dominio de técnicas de organización poderosas aceleran nuestra capacidad para crear programas grandes y significativos. A la inversa, dado que escribir programas grandes es muy exigente, nos vemos estimulados a inventar nuevos métodos para reducir la masa de funciones y detalles que deben ajustarse en los programas grandes.

A diferencia de los programas, los ordenadores deben obedecer las leyes de la física. Si desean funcionar rápidamente —unos pocos nanosegundos por cambio de estado—, deben transmitir electrones solo a pequeñas distancias (como máximo $1 {1\over2}$ pies). El calor generado por el enorme número de dispositivos tan concentrados en el espacio tiene que ser eliminado. Se ha desarrollado un exquisito arte de ingeniería que equilibra la multiplicidad de funciones y la densidad de dispositivos. En cualquier caso, el hardware siempre opera a un nivel más primitivo que aquel en el que nos interesa programar. Los procesos que transforman nuestros programas Lisp en programas de “máquina” son en sí mismos modelos abstractos que programamos. Su estudio y creación dan una gran cantidad de conocimiento sobre los programas de organización asociados con la programación de modelos arbitrarios. Por supuesto, el propio ordenador puede ser modelado de esa manera. Piénselo: ¡el comportamiento del elemento de conmutación físico más pequeño es modelado por la mecánica cuántica descrita por ecuaciones diferenciales cuyo comportamiento detallado es capturado por aproximaciones numéricas representadas en programas de ordenador que se

ejecutan en ordenadores compuestos de…!

No es simplemente una cuestión de conveniencia táctica identificar por separado los tres focos. Aunque, como dicen, todo está en la cabeza, esta separación lógica induce una aceleración del tráfico simbólico entre estos focos cuya riqueza, vitalidad y potencial solo es superada en la experiencia humana por la evolución de la vida misma. En el mejor de los casos, las relaciones entre los focos son metaestables. Los ordenadores nunca son lo suficientemente grandes ni lo suficientemente rápidos. Cada avance en la tecnología de hardware conduce a empresas de programación más masivas, nuevos principios de organización y un enriquecimiento de los modelos abstractos. Cada lector debería preguntarse periódicamente "¿Hacia qué fin, hacia qué fin?" —pero no se lo pregunte con demasiada frecuencia, no sea que deje pasar la diversión de la programación por el estreñimiento de la filosofía agridulce.

Entre los programas que escribimos, algunos (pero nunca suficientes) realizan una función matemática precisa como ordenar o encontrar el máximo de una secuencia de números, determinar la primalidad o encontrar la raíz cuadrada. A tales programas los llamamos algoritmos, y se sabe mucho de su comportamiento óptimo, particularmente con respecto a los dos importantes parámetros del tiempo de ejecución y los requisitos de almacenamiento de datos. Un programador debe adquirir buenos algoritmos y modismos. Aunque algunos programas se resisten a especificaciones precisas, es responsabilidad del programador estimar, y siempre intentar mejorar, su rendimiento.

Lisp es un superviviente, habiendo estado en uso durante aproximadamente un cuarto de siglo. Entre los lenguajes de programación activos, solo Fortran ha tenido una vida más larga. Ambos lenguajes han apoyado las necesidades de programación de importantes áreas de aplicación, Fortran para la computación científica y de ingeniería y Lisp para la inteligencia artificial. Estas dos áreas continúan siendo importantes, y sus programadores están tan dedicados a estos dos lenguajes que Lisp y Fortran bien podrían continuar en uso activo durante al menos otro cuarto de siglo.

Lisp cambia. El dialecto Scheme utilizado en este texto ha evolucionado desde el Lisp original y difiere de este último en varias formas importantes, incluyendo el alcance estático para la vinculación de variables y permitir que las funciones devuelvan funciones como valores. En su estructura semántica, Scheme está tan estrechamente emparentado con Algol 60 como con los primeros Lisps. Algol 60, que nunca volverá a ser un lenguaje activo, vive en los genes de Scheme y Pascal. Sería difícil encontrar dos lenguajes que sean la moneda de cambio de dos culturas más diferentes que las reunidas en torno a estos dos lenguajes. Pascal es para construir pirámides: estructuras

imponentes, impresionantes y estáticas construidas por ejércitos que empujan pesados bloques a su lugar. Lisp es para construir organismos: estructuras imponentes, impresionantes y dinámicas construidas por escuadrones que ajustan miríadas fluctuantes de organismos más simples en su lugar. Los principios organizativos utilizados son los mismos en ambos casos, excepto por una diferencia extraordinariamente importante: la funcionalidad discrecional exportable confiada al programador individual de Lisp es más de un orden de magnitud mayor que la que se encuentra en las empresas de Pascal. Los programas de Lisp inflan las bibliotecas con funciones cuya utilidad trasciende la aplicación que las produjo. La lista, la estructura de datos nativa de Lisp, es en gran parte responsable de tal crecimiento de la utilidad. La estructura simple y la aplicabilidad natural de las listas se reflejan en funciones que son sorprendentemente no idiosincrásicas. En Pascal, la plétora de estructuras de datos declarables induce una especialización dentro de las funciones que inhibe y penaliza la cooperación casual. Es mejor tener 100 funciones que operen sobre una estructura de datos que tener 10 funciones que operen sobre 10 estructuras de datos. Como resultado, la pirámide debe permanecer sin cambios durante un milenio; el organismo debe evolucionar o perecer.

Para ilustrar esta diferencia, compare el tratamiento del material y los ejercicios de este libro con el de cualquier texto de primer curso que utilice Pascal. No se engañe con la ilusión de que este es un texto digerible solo en el MIT, peculiar de la raza que se encuentra allí. Es precisamente lo que un libro serio sobre la programación en Lisp debe ser, sin importar quién sea el estudiante o dónde se utilice.

Tenga en cuenta que este es un texto sobre programación, a diferencia de la mayoría de los libros de Lisp, que se utilizan como preparación para el trabajo en inteligencia artificial. Después de todo, las preocupaciones críticas de programación de la ingeniería de software y la inteligencia artificial tienden a converger a medida que los sistemas bajo investigación se hacen más grandes. Esto explica por qué hay un interés creciente en Lisp fuera de la inteligencia artificial.

Como era de esperar de sus objetivos, la investigación en inteligencia artificial genera muchos problemas de programación significativos. En otras culturas de programación, esta avalancha de problemas genera nuevos lenguajes. De hecho, en cualquier tarea de programación muy grande, un principio organizativo útil es controlar y aislar el tráfico dentro de los módulos de la tarea mediante la invención de un lenguaje. Estos lenguajes tienden a volverse menos primitivos a medida que uno se acerca a los límites del sistema donde los humanos interactuamos con mayor frecuencia. Como resultado, tales sistemas contienen funciones complejas de procesamiento de

lenguaje replicadas muchas veces. Lisp tiene una sintaxis y una semántica tan simples que el análisis sintáctico puede tratarse como una tarea elemental. Así, la tecnología de análisis sintáctico casi no juega ningún papel en los programas de Lisp, y la construcción de procesadores de lenguaje rara vez es un impedimento para la tasa de crecimiento y cambio de los grandes sistemas de Lisp. Finalmente, es esta misma simplicidad de sintaxis y semántica la responsable de la carga y la libertad que soportan todos los programadores de Lisp. Ningún programa de Lisp de ningún tamaño más allá de unas pocas líneas puede escribirse sin estar saturado de funciones discrecionales. ¡Inventar y encajar; tener ataques y reinventar! Brindamos por el programador de Lisp que escribe sus pensamientos dentro de nidos de paréntesis.

Alan J. Perlis
New Haven, Connecticut

Prefacio a la segunda edición

> ¿Es posible que el software no sea como cualquier otra cosa, que esté destinado a ser descartado: que el punto principal sea verlo siempre como una pompa de jabón?
>
> —Alan J. Perlis

El material de este libro ha sido la base de la asignatura de nivel de entrada de informática del MIT desde 1980. Habíamos estado enseñando este material durante cuatro años cuando se publicó la primera edición, y han transcurrido doce años más hasta la aparición de esta segunda edición. Nos complace que nuestro trabajo haya sido ampliamente adoptado e incorporado en otros textos. Hemos visto a nuestros estudiantes tomar las ideas y los programas de este libro y construirlos como el núcleo de nuevos sistemas y lenguajes informáticos. En una realización literal de un antiguo juego de palabras talmúdico, nuestros estudiantes se han convertido en nuestros constructores. Tenemos la suerte de tener estudiantes tan capaces y constructores tan consumados.

Al preparar esta edición, hemos incorporado cientos de aclaraciones sugeridas por nuestra propia experiencia docente y los comentarios de colegas en el MIT y en otros lugares. Hemos rediseñado la mayoría de los principales sistemas de programación del libro, incluido el sistema de aritmética genérica, los intérpretes, el simulador de máquina de registros y el compilador; y hemos reescrito todos los ejemplos de programas para asegurar que cualquier implementación de Scheme que se ajuste al estándar de Scheme del IEEE (IEEE 1990) pueda ejecutar el código.

Esta edición enfatiza varios temas nuevos. El más importante de estos es el papel central que juegan los diferentes enfoques para tratar el tiempo en los modelos computacionales: objetos con estado, programación concurrente, programación funcional, evaluación diferida y programación no determinista. Hemos incluido nuevas secciones sobre concurrencia y no determinismo, y hemos intentado integrar este tema en todo el libro.

La primera edición del libro seguía de cerca el programa de nuestra asignatura de un semestre en el MIT. Con todo el nuevo material de la segunda edición, no será posible cubrir todo en un solo semestre, por lo que el instructor tendrá que elegir. En nuestra propia enseñanza, a veces omitimos la sección sobre programación lógica (Section 4.4), hacemos que los estudiantes usen el simulador de máquina de registros pero no cubrimos su implementación (Section 5.2), y damos solo una visión superficial del compilador (Section 5.5). Aun así, sigue siendo un curso intenso. Algunos instructores pueden desear cubrir solo los primeros tres o cuatro capítulos, dejando el otro material para cursos posteriores.

El sitio web http://mitpress.mit.edu/sicp proporciona soporte para los usuarios de este libro. Esto incluye programas del libro, ejemplos de tareas de programación, materiales complementarios e implementaciones descargables del dialecto Scheme de Lisp.

Prefacio a la primera edición

> Un ordenador es como un violín. Puedes imaginar a un novato probando primero un fonógrafo y luego un violín. Este último, dice, suena terrible. Ese es el argumento que hemos oído de nuestros humanistas y de la mayoría de nuestros informáticos. Los programas de ordenador son buenos, dicen, para propósitos particulares, pero no son flexibles. Tampoco lo es un violín, o una máquina de escribir, hasta que aprendes a usarlo.
>
> —Marvin Minsky, "Por qué la programación es un buen medio para expresar ideas mal entendidas y formuladas descuidadamente"

"Estructura e Interpretación de Programas de Computadoras" es la asignatura de nivel de entrada en informática en el Instituto de Tecnología de Massachusetts. Es obligatoria para todos los estudiantes del MIT que se especializan en ingeniería eléctrica o en informática, como una cuarta parte del "currículo común básico", que también incluye dos asignaturas sobre circuitos y sistemas lineales y una asignatura sobre el diseño de sistemas digitales. Hemos estado involucrados en el desarrollo de esta asignatura desde 1978, y hemos enseñado este material en su forma actual desde el otoño de 1980 a entre 600 y 700 estudiantes cada año. La mayoría de estos estudiantes han tenido poca o ninguna formación formal previa en computación, aunque muchos han jugado un poco con los ordenadores y algunos han tenido una amplia experiencia en programación o diseño de hardware.

Nuestro diseño de esta asignatura introductoria de informática refleja dos preocupaciones principales. Primero, queremos establecer la idea de que un lenguaje de ordenador no es solo una forma de hacer que un ordenador realice operaciones, sino que es un medio formal novedoso para expresar ideas sobre metodología. Así, los programas deben escribirse para que la gente los lea, y solo incidentalmente para que las máquinas los ejecuten. Segundo, creemos que el material esencial que debe abordarse en una asignatura a este nivel no es la sintaxis de construcciones particulares de un lenguaje de programación, ni algoritmos ingeniosos para calcular funciones

particulares de manera eficiente, ni siquiera el análisis matemático de algoritmos y los fundamentos de la computación, sino más bien las técnicas utilizadas para controlar la complejidad intelectual de los grandes sistemas de software.

Nuestro objetivo es que los estudiantes que completen esta asignatura tengan una buena idea de los elementos de estilo y la estética de la programación. Deben dominar las principales técnicas para controlar la complejidad en un sistema grande. Deben ser capaces de leer un programa de 50 páginas, si está escrito en un estilo ejemplar. Deben saber qué no leer y qué no necesitan entender en un momento dado. Deben sentirse seguros al modificar un programa, manteniendo el espíritu y el estilo del autor original.

Estas habilidades no son en absoluto únicas de la programación de ordenadores. Las técnicas que enseñamos y de las que nos basamos son comunes a todo el diseño de ingeniería. Controlamos la complejidad construyendo abstracciones que ocultan detalles cuando es apropiado. Controlamos la complejidad estableciendo interfaces convencionales que nos permiten construir sistemas combinando piezas estándar y bien entendidas de una manera de "mezclar y combinar". Controlamos la complejidad estableciendo nuevos lenguajes para describir un diseño, cada uno de los cuales enfatiza aspectos particulares del diseño y resta importancia a otros.

Subyacente a nuestro enfoque de esta asignatura está nuestra convicción de que la "informática" no es una ciencia y que su importancia tiene poco que ver con los ordenadores. La revolución informática es una revolución en la forma en que pensamos y en la forma en que expresamos lo que pensamos. La esencia de este cambio es la aparición de lo que podría llamarse mejor *epistemología procedimental* —el estudio de la estructura del conocimiento desde un punto de vista imperativo, en oposición al punto de vista más declarativo adoptado por las asignaturas matemáticas clásicas. Las matemáticas proporcionan un marco para tratar con precisión las nociones de "qué es". La computación proporciona un marco para tratar con precisión las nociones de "cómo hacer".

En la enseñanza de nuestro material utilizamos un dialecto del lenguaje de programación Lisp. Nunca enseñamos formalmente el lenguaje, porque no es necesario. Simplemente lo usamos, y los estudiantes lo aprenden en unos pocos días. Esta es una gran ventaja de los lenguajes tipo Lisp: tienen muy pocas formas de formar expresiones compuestas y casi ninguna estructura sintáctica. Todas las propiedades formales se pueden cubrir en una hora, como las reglas del ajedrez. Después de un corto tiempo, nos olvidamos de los detalles sintácticos del lenguaje (porque no hay ninguno) y nos ocupamos de los problemas reales: averiguar qué queremos calcular, cómo

descompondremos los problemas en partes manejables y cómo trabajaremos en las partes. Otra ventaja de Lisp es que soporta (pero no impone) más de las estrategias a gran escala para la descomposición modular de programas que cualquier otro lenguaje que conozcamos. Podemos hacer abstracciones procedimentales y de datos, podemos usar funciones de orden superior para capturar patrones comunes de uso, podemos modelar el estado local usando asignación y mutación de datos, podemos vincular partes de un programa con flujos y evaluación diferida, y podemos implementar fácilmente lenguajes incrustados. Todo esto está incrustado en un entorno interactivo con un excelente soporte para el diseño, construcción, prueba y depuración incremental de programas. Agradecemos a todas las generaciones de magos de Lisp, comenzando con John McCarthy, que han creado una fina herramienta de poder y elegancia sin precedentes.

Scheme, el dialecto de Lisp que usamos, es un intento de reunir el poder y la elegancia de Lisp y Algol. De Lisp tomamos el poder metalingüístico que se deriva de la sintaxis simple, la representación uniforme de programas como objetos de datos y los datos asignados en el montón con recolección de basura. De Algol tomamos el alcance léxico y la estructura de bloques, que son regalos de los pioneros del diseño de lenguajes de programación que estaban en el comité de Algol. Deseamos citar a John Reynolds y Peter Landin por sus ideas sobre la relación del cálculo λ de Church con la estructura de los lenguajes de programación. También reconocemos nuestra deuda con los matemáticos que exploraron este territorio décadas antes de que aparecieran los ordenadores. Estos pioneros incluyen a Alonzo Church, Barkley Rosser, Stephen Kleene y Haskell Curry.

Agradecimientos

NOS GUSTARÍA AGRADECER a las muchas personas que nos han ayudado a desarrollar este libro y este currículo.

Nuestra asignatura es un claro descendiente intelectual de “6.231”, una maravillosa asignatura sobre lingüística de la programación y el cálculo λ impartida en el MIT a finales de la década de 1960 por Jack Wozencraft y Arthur Evans, Jr.

Tenemos una gran deuda con Robert Fano, quien reorganizó el currículo introductorio de ingeniería eléctrica e informática del MIT para enfatizar los principios del diseño de ingeniería. Él nos guió al comenzar esta empresa y escribió el primer conjunto de notas de la asignatura a partir de las cuales evolucionó este libro.

Gran parte del estilo y la estética de la programación que intentamos enseñar se desarrolló en conjunto con Guy Lewis Steele Jr., quien colaboró con Gerald Jay Sussman en el desarrollo inicial del lenguaje Scheme. Además, David Turner, Peter Henderson, Dan Friedman, David Wise y Will Clinger nos han enseñado muchas de las técnicas de la comunidad de programación funcional que aparecen en este libro.

Joel Moses nos enseñó sobre la estructuración de grandes sistemas. Su experiencia con el sistema Macsyma para la computación simbólica proporcionó la idea de que se deben evitar las complejidades del control y concentrarse en organizar los datos para reflejar la estructura real del mundo que se está modelando.

Marvin Minsky y Seymour Papert formaron muchas de nuestras actitudes sobre la programación y su lugar en nuestras vidas intelectuales. A ellos les debemos la comprensión de que la computación proporciona un medio de expresión para explorar ideas que de otro modo serían demasiado complejas para tratar con precisión. Ellos enfatizan que la capacidad de un estudiante para escribir y modificar programas proporciona un medio poderoso en el que la exploración se convierte en una actividad natural.

También estamos totalmente de acuerdo con Alan Perlis en que la programación es muy divertida y es mejor que tengamos cuidado de apoyar la alegría de programar. Parte de esta alegría se deriva de observar a grandes maestros en acción. Somos afortunados de haber sido aprendices de programadores a los pies de Bill Gosper y Richard Greenblatt.

Es difícil identificar a todas las personas que han contribuido al desarrollo de nuestro currículo. Agradecemos a todos los profesores, instructores de recitación y tutores que han trabajado con nosotros durante los últimos quince años y han dedicado muchas horas extra a nuestra asignatura, especialmente a Bill Siebert, Albert Meyer, Joe Stoy, Randy Davis, Louis Braida, Eric Grimson, Rod Brooks, Lynn Stein y Peter Szolovits. Nos gustaría reconocer especialmente las destacadas contribuciones docentes de Franklyn Turbak, ahora en Wellesley; su trabajo en la instrucción de pregrado estableció un estándar al que todos podemos aspirar. Agradecemos a Jerry Saltzer y Jim Miller por ayudarnos a lidiar con los misterios de la concurrencia, y a Peter Szolovits y David McAllester por sus contribuciones a la exposición de la evaluación no determinista en el Capítulo 4.

Muchas personas han realizado un esfuerzo significativo presentando este material en otras universidades. Algunas de las personas con las que hemos trabajado de cerca son Jacob Katzenelson en el Technion, Hardy Mayer en la Universidad de California en Irvine, Joe Stoy en Oxford, Elisha Sacks en Purdue y Jan Komorowski en la Universidad Noruega de Ciencia y Tecnología. Estamos excepcionalmente orgullosos de nuestros colegas que han recibido importantes premios de enseñanza por sus adaptaciones de esta asignatura en otras universidades, incluidos Kenneth Yip en Yale, Brian Harvey en la Universidad de California en Berkeley y Dan Huttenlocher en Cornell.

Al Moyé organizó que enseñáramos este material a ingenieros de Hewlett-Packard y la producción de cintas de video de estas conferencias. Nos gustaría agradecer a los talentosos instructores, en particular a Jim Miller, Bill Siebert y Mike Eisenberg, que han diseñado cursos de educación continua que incorporan estas cintas y los han impartido en universidades e industrias de todo el mundo.

Muchos educadores en otros países han realizado un trabajo significativo traduciendo la primera edición. Michel Briand, Pierre Chamard y André Pic produjeron una edición en francés; Susanne Daniels-Herold produjo una edición en alemán; y Fumio Motoyoshi produjo una edición en japonés. No sabemos quién produjo la edición en chino, pero consideramos un honor haber sido seleccionados como objeto de una traducción “no autorizada”.

Es difícil enumerar a todas las personas que han realizado contribuciones técnicas al desarrollo de los sistemas Scheme que utilizamos con fines educativos. Además de Guy Steele, los magos principales han incluido a Chris Hanson, Joe Bowbeer, Jim Miller, Guillermo Rozas y Stephen Adams. Otros que han dedicado un tiempo significativo son Richard Stallman, Alan Bawden, Kent Pitman, Jon Taft, Neil Mayle, John Lamping, Gwyn Osnos, Tracy Larrabee, George Carrette, Soma Chaudhuri, Bill Chiarchiaro, Steven Kirsch, Leigh Klotz, Wayne Noss, Todd Cass, Patrick O'Donnell, Kevin Theobald, Daniel Weise, Kenneth Sinclair, Anthony Courtemanche, Henry M. Wu, Andrew Berlin y Ruth Shyu.

Más allá de la implementación del MIT, nos gustaría agradecer a las muchas personas que trabajaron en el estándar de Scheme del IEEE, incluidos William Clinger y Jonathan Rees, que editaron el $\rm Rˆ4RS$, y Chris Haynes, David Bartley, Chris Hanson y Jim Miller, que prepararon el estándar del IEEE.

Dan Friedman ha sido un líder de larga data de la comunidad Scheme. El trabajo más amplio de la comunidad va más allá de los problemas de diseño del lenguaje para abarcar innovaciones educativas significativas, como el currículo de secundaria basado en EdScheme de Schemer's Inc., y los maravillosos libros de Mike Eisenberg y de Brian Harvey y Matthew Wright.

Apreciamos el trabajo de aquellos que contribuyeron a hacer de este un libro real, especialmente Terry Ehling, Larry Cohen y Paul Bethge en MIT Press. Ella Mazel encontró la maravillosa imagen de la portada. Para la segunda edición, estamos particularmente agradecidos a Bernard y Ella Mazel por su ayuda con el diseño del libro, y a David Jones, mago extraordinario de TeX. También estamos en deuda con aquellos lectores que hicieron comentarios penetrantes sobre el nuevo borrador: Jacob Katzenelson, Hardy Mayer, Jim Miller y especialmente Brian Harvey, quien hizo con este libro lo que Julie hizo con su libro *Simply Scheme*.

Finalmente, nos gustaría agradecer el apoyo de las organizaciones que han fomentado este trabajo a lo largo de los años, incluido el apoyo de Hewlett-Packard, hecho posible por Ira Goldstein y Joel Birnbaum, y el apoyo de DARPA, hecho posible por Bob Kahn.

Construcción de abstracciones con procedimientos

> Los actos de la mente, en los que ejerce su poder sobre las ideas simples, son principalmente estos tres: 1. Combinar varias ideas simples en una compuesta, y así se hacen todas las ideas complejas. 2. El segundo es juntar dos ideas, ya sean simples o complejas, y ponerlas una al lado de la otra para verlas a la vez, sin unirlas en una, con lo que obtiene todas sus ideas de relaciones. 3. El tercero es separarlas de todas las demás ideas que las acompañan en su existencia real: esto se llama abstracción, y así se hacen todas sus ideas generales.
>
> —John Locke, *Ensayo sobre el entendimiento humano* (1690)

ESTAMOS A PUNTO DE ESTUDIAR la idea de un *proceso computacional*. Los procesos computacionales son seres abstractos que habitan en los ordenadores. A medida que evolucionan, los procesos manipulan otras cosas abstractas llamadas *datos*. La evolución de un proceso está dirigida por un patrón de reglas llamado *programa*. La gente crea programas para dirigir procesos. En efecto, conjuramos los espíritus del ordenador con nuestros hechizos.

Un proceso computacional es, de hecho, muy parecido a la idea que un hechicero tiene de un espíritu. No se puede ver ni tocar. No está compuesto de materia en absoluto. Sin embargo, es muy real. Puede realizar trabajo intelectual. Puede responder preguntas. Puede afectar al mundo desembolsando dinero en un banco o controlando un brazo robótico en una fábrica. Los programas que usamos para conjurar procesos son como los hechizos de un hechicero. Están cuidadosamente compuestos de expresiones simbólicas en *lenguajes de programación* arcanos y esotéricos que prescriben las tareas que queremos que realicen nuestros procesos.

Un proceso computacional, en un ordenador que funciona correctamente, ejecuta programas con precisión y exactitud. Así, como el aprendiz de hechicero, los programadores novatos deben aprender a comprender y anticipar las consecuencias de sus conjuros. Incluso los pequeños errores

(generalmente llamados *bugs* o *glitches*) en los programas pueden tener consecuencias complejas e imprevistas.

Afortunadamente, aprender a programar es considerablemente menos peligroso que aprender hechicería, porque los espíritus con los que tratamos están convenientemente contenidos de una manera segura. La programación en el mundo real, sin embargo, requiere cuidado, experiencia y sabiduría. Un pequeño error en un programa de diseño asistido por ordenador, por ejemplo, puede llevar al colapso catastrófico de un avión o una presa o a la autodestrucción de un robot industrial.

Los ingenieros de software maestros tienen la capacidad de organizar programas para que puedan estar razonablemente seguros de que los procesos resultantes realizarán las tareas previstas. Pueden visualizar el comportamiento de sus sistemas por adelantado. Saben cómo estructurar programas para que los problemas imprevistos no conduzcan a consecuencias catastróficas, y cuando surgen problemas, pueden *depurar* sus programas. Los sistemas computacionales bien diseñados, como los automóviles o los reactores nucleares bien diseñados, se diseñan de manera modular, de modo que las partes se puedan construir, reemplazar y depurar por separado.

Programación en Lisp

Necesitamos un lenguaje apropiado para describir procesos, y para este propósito usaremos el lenguaje de programación Lisp. Así como nuestros pensamientos cotidianos se expresan generalmente en nuestro lenguaje natural (como inglés, francés o japonés), y las descripciones de fenómenos cuantitativos se expresan con notaciones matemáticas, nuestros pensamientos procedimentales se expresarán en Lisp. Lisp fue inventado a finales de la década de 1950 como un formalismo para razonar sobre el uso de ciertos tipos de expresiones lógicas, llamadas *ecuaciones de recursión*, como modelo de computación. El lenguaje fue concebido por John McCarthy y se basa en su artículo “Funciones recursivas de expresiones simbólicas y su computación por máquina” (McCarthy 1960).

A pesar de su concepción como un formalismo matemático, Lisp es un lenguaje de programación práctico. Un *intérprete* de Lisp es una máquina que lleva a cabo procesos descritos en el lenguaje Lisp. El primer intérprete de Lisp fue implementado por McCarthy con la ayuda de colegas y estudiantes en el Grupo de Inteligencia Artificial del Laboratorio de Investigación de Electrónica del MIT y en el Centro de Computación del MIT.[1] Lisp, cuyo nombre es un acrónimo de Procesamiento de LIStas, fue diseñado para proporcionar capacidades de manipulación de símbolos para abordar problemas de programación como la diferenciación e integración simbólica

de expresiones algebraicas. Incluía para este propósito nuevos objetos de datos conocidos como átomos y listas, lo que lo distinguía notablemente de todos los demás lenguajes de la época.

Lisp no fue el producto de un esfuerzo de diseño concertado. En cambio, evolucionó informalmente de manera experimental en respuesta a las necesidades de los usuarios y a consideraciones pragmáticas de implementación. La evolución informal de Lisp ha continuado a lo largo de los años, y la comunidad de usuarios de Lisp tradicionalmente se ha resistido a los intentos de promulgar cualquier definición "oficial" del lenguaje. Esta evolución, junto con la flexibilidad y elegancia de la concepción inicial, ha permitido que Lisp, que es el segundo lenguaje más antiguo en uso generalizado hoy en día (solo Fortran es más antiguo), se adapte continuamente para abarcar las ideas más modernas sobre el diseño de programas. Así, Lisp es ahora una familia de dialectos que, aunque comparten la mayoría de las características originales, pueden diferir entre sí de maneras significativas. El dialecto de Lisp utilizado en este libro se llama Scheme.[2]

Debido a su carácter experimental y su énfasis en la manipulación de símbolos, Lisp fue al principio muy ineficiente para los cálculos numéricos, al menos en comparación con Fortran. Sin embargo, con los años se han desarrollado compiladores de Lisp que traducen los programas a código máquina que puede realizar cálculos numéricos de manera razonablemente eficiente. Y para aplicaciones especiales, Lisp se ha utilizado con gran eficacia.[3] Aunque Lisp aún no ha superado su antigua reputación de ser irremediablemente ineficiente, ahora se utiliza en muchas aplicaciones donde la eficiencia no es la principal preocupación. Por ejemplo, Lisp se ha convertido en un lenguaje de elección para los lenguajes de shell de sistemas operativos y para los lenguajes de extensión de editores y sistemas de diseño asistido por ordenador.

Si Lisp no es un lenguaje convencional, ¿por qué lo usamos como marco para nuestra discusión sobre la programación? Porque el lenguaje posee características únicas que lo convierten en un excelente medio para estudiar construcciones de programación y estructuras de datos importantes y para relacionarlas con las características lingüísticas que las soportan. La más significativa de estas características es el hecho de que las descripciones de procesos en Lisp, llamadas *procedimientos*, pueden representarse y manipularse como datos de Lisp. La importancia de esto es que existen poderosas técnicas de diseño de programas que se basan en la capacidad de difuminar la distinción tradicional entre datos "pasivos" y procesos "activos". Como descubriremos, la flexibilidad de Lisp en el manejo de procedimientos como datos lo convierte en uno de los lenguajes más convenientes que

existen para explorar estas técnicas. La capacidad de representar procedimientos como datos también convierte a Lisp en un excelente lenguaje para escribir programas que deben manipular otros programas como datos, como los intérpretes y compiladores que soportan los lenguajes informáticos. Además de estas consideraciones, programar en Lisp es muy divertido.

Los elementos de la programación

Un lenguaje de programación potente es más que un simple medio para instruir a un ordenador a realizar tareas. El lenguaje también sirve como un marco dentro del cual organizamos nuestras ideas sobre los procesos. Por lo tanto, cuando describimos un lenguaje, debemos prestar especial atención a los medios que el lenguaje proporciona para combinar ideas simples para formar ideas más complejas. Todo lenguaje potente tiene tres mecanismos para lograr esto:

- **expresiones primitivas**, que representan las entidades más simples con las que el lenguaje se ocupa,
- **medios de combinación**, mediante los cuales se construyen elementos compuestos a partir de otros más simples, y
- **medios de abstracción**, mediante los cuales los elementos compuestos pueden ser nombrados y manipulados como unidades.

En programación, tratamos con dos tipos de elementos: procedimientos y datos. (Más adelante descubriremos que en realidad no son tan distintos). Informalmente, los datos son “cosas$\kern0.1em$” que queremos manipular, y los procedimientos son descripciones de las reglas para manipular los datos. Por lo tanto, cualquier lenguaje de programación potente debe ser capaz de describir datos primitivos y procedimientos primitivos y debe tener métodos para combinar y abstraer procedimientos y datos.

En este capítulo solo trataremos con datos numéricos simples para que podamos centrarnos en las reglas para construir procedimientos.[4] En capítulos posteriores veremos que estas mismas reglas nos permiten construir procedimientos para manipular también datos compuestos.

Expresiones

Una manera fácil de empezar a programar es examinar algunas interacciones

típicas con un intérprete para el dialecto Scheme de Lisp. Imagina que estás sentado en una terminal de ordenador. Escribes una *expresión*, y el intérprete responde mostrando el resultado de su *evaluación* de esa expresión.

Un tipo de expresión primitiva que podrías escribir es un número. (Más precisamente, la expresión que escribes consiste en los numerales que representan el número en base 10.) Si le presentas a Lisp un número

```
486
```

el intérprete responderá imprimiendo[5]

```
486
```

Las expresiones que representan números pueden combinarse con una expresión que representa un procedimiento primitivo (como `+` o `*`) para formar una expresión compuesta que representa la aplicación del procedimiento a esos números. Por ejemplo:

`(+ 137 349)` *486*

`(- 1000 334)` *666*

`(* 5 99)` *495*

`(/ 10 5)` *2*

`(+ 2.7 10)` *12.7*

Expresiones como estas, formadas al delimitar una lista de expresiones entre paréntesis para denotar la aplicación de un procedimiento, se llaman *combinaciones*. El elemento más a la izquierda de la lista se llama *operador*, y los otros elementos se llaman *operandos*. El valor de una combinación se obtiene aplicando el procedimiento especificado por el operador a los *argumentos* que son los valores de los operandos.

La convención de colocar el operador a la izquierda de los operandos se conoce como *notación prefija*, y puede ser algo confusa al principio porque se aparta significativamente de la convención matemática habitual. La notación prefija tiene varias ventajas, sin embargo. Una de ellas es que puede acomodar procedimientos que pueden tomar un número arbitrario de argumentos, como en los siguientes ejemplos:

`(+ 21 35 12 7)` *75*

`(* 25 4 12)` *1200*

No puede surgir ambigüedad, porque el operador siempre es el elemento más a la izquierda y toda la combinación está delimitada por los paréntesis.

Una segunda ventaja de la notación prefija es que se extiende de manera sencilla para permitir que las combinaciones se *aniden*, es decir, que tengan combinaciones cuyos elementos son a su vez combinaciones:

```
(+ (* 3 5) (- 10 6))  19
```

No hay límite (en principio) para la profundidad de dicha anidación y para la complejidad general de las expresiones que el intérprete de Lisp puede evaluar. Somos nosotros, los humanos, los que nos confundimos con expresiones aún relativamente simples como

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
```

que el intérprete evaluaría fácilmente como 57. Podemos ayudarnos escribiendo tal expresión en la forma

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
```

siguiendo una convención de formato conocida como *pretty-printing*, en la que cada combinación larga se escribe de modo que los operandos estén alineados verticalmente. Las indentaciones resultantes muestran claramente la estructura de la expresión.[6]

Incluso con expresiones complejas, el intérprete siempre opera en el mismo ciclo básico: lee una expresión de la terminal, evalúa la expresión e imprime el resultado. Este modo de operación a menudo se expresa diciendo que el intérprete se ejecuta en un *bucle de lectura-evaluación-impresión*. Observe en particular que no es necesario instruir explícitamente al intérprete para que imprima el valor de la expresión.[7]

Nombres y el entorno

Un aspecto crítico de un lenguaje de programación es el medio que proporciona para usar nombres para referirse a objetos computacionales. Decimos que el nombre identifica una *variable* cuyo *valor* es el objeto.

En el dialecto Scheme de Lisp, nombramos las cosas con `define`. Escribir

```
(define size 2)
```

hace que el intérprete asocie el valor 2 con el nombre `size`.[8] Una vez que el nombre `size` ha sido asociado con el número 2, podemos referirnos al valor

2 por su nombre:

size *2*

(* 5 size) *10*

Aquí hay más ejemplos del uso de `define`:

(define pi 3.14159) (define radius 10) (* pi (* radius radius)) *314.159*
(define circumference (* 2 pi radius)) circumference *62.8318*

`define` es el medio de abstracción más simple de nuestro lenguaje, ya que nos permite usar nombres simples para referirnos a los resultados de operaciones compuestas, como la `circumference` calculada anteriormente. En general, los objetos computacionales pueden tener estructuras muy complejas, y sería extremadamente inconveniente tener que recordar y repetir sus detalles cada vez que queremos usarlos. De hecho, los programas complejos se construyen construyendo, paso a paso, objetos computacionales de complejidad creciente. El intérprete hace que esta construcción de programas paso a paso sea particularmente conveniente porque las asociaciones nombre-objeto se pueden crear incrementalmente en interacciones sucesivas. Esta característica fomenta el desarrollo y la prueba incremental de programas y es en gran parte responsable del hecho de que un programa Lisp generalmente consta de un gran número de procedimientos relativamente simples.

Debería estar claro que la posibilidad de asociar valores con símbolos y luego recuperarlos significa que el intérprete debe mantener algún tipo de memoria que haga un seguimiento de los pares nombre-objeto. Esta memoria se llama el *entorno* (más precisamente, el *entorno global*, ya que veremos más adelante que una computación puede involucrar varios entornos diferentes).[9]

Evaluación de combinaciones

Uno de nuestros objetivos en este capítulo es aislar problemas sobre el pensamiento procedimental. Como ejemplo, consideremos que, al evaluar combinaciones, el propio intérprete sigue un procedimiento.

Para evaluar una combinación, haga lo siguiente:

1. Evalúe las subexpresiones de la combinación.

2. Aplique el procedimiento que es el valor de la subexpresión más a la izquierda (el operador) a los argumentos que son los valores de las otras

subexpresiones (los operandos).

Incluso esta simple regla ilustra algunos puntos importantes sobre los procesos en general. Primero, observe que el primer paso dicta que para llevar a cabo el proceso de evaluación de una combinación, primero debemos realizar el proceso de evaluación en cada elemento de la combinación. Por lo tanto, la regla de evaluación es de naturaleza *recursiva*; es decir, incluye, como uno de sus pasos, la necesidad de invocar la propia regla.[10]

Observe cuán sucintamente se puede usar la idea de recursión para expresar lo que, en el caso de una combinación profundamente anidada, de otro modo se vería como un proceso bastante complicado. Por ejemplo, evaluar

```
(* (+ 2 (* 4 6)) (+ 3 5 7))
```

requiere que la regla de evaluación se aplique a cuatro combinaciones diferentes. Podemos obtener una imagen de este proceso representando la combinación en forma de árbol, como se muestra en la Figura 1.1. Cada combinación está representada por un nodo con ramas correspondientes al operador y los operandos de la combinación que parten de él. Los nodos terminales (es decir, nodos sin ramas que parten de ellos) representan operadores o números. Viendo la evaluación en términos del árbol, podemos imaginar que los valores de los operandos se filtran hacia arriba, comenzando desde los nodos terminales y luego combinándose en niveles cada vez más altos. En general, veremos que la recursión es una técnica muy poderosa para tratar con objetos jerárquicos, tipo árbol. De hecho, la forma de la regla de evaluación de "filtrar valores hacia arriba" es un ejemplo de un tipo general de proceso conocido como *acumulación de árbol*.

Figura 1.1: Representación en árbol, que muestra el valor de cada subcombinación.

A continuación, observe que la aplicación repetida del primer paso nos lleva al punto en que necesitamos evaluar, no combinaciones, sino expresiones primitivas como numerales, operadores incorporados u otros nombres. Nos ocupamos de los casos primitivos estipulando que

- los valores de los numerales son los números que nombran,
- los valores de los operadores incorporados son las secuencias de instrucciones de máquina que llevan a cabo las operaciones correspondientes, y

- los valores de otros nombres son los objetos asociados con esos nombres en el entorno.

Podemos considerar la segunda regla como un caso especial de la tercera estipulando que símbolos como `+` y `*` también están incluidos en el entorno global, y están asociados con las secuencias de instrucciones de máquina que son sus "valores". El punto clave a notar es el papel del entorno en la determinación del significado de los símbolos en las expresiones. En un lenguaje interactivo como Lisp, no tiene sentido hablar del valor de una expresión como `(+ x 1)` sin especificar ninguna información sobre el entorno que proporcionaría un significado para el símbolo `x` (o incluso para el símbolo `+`). Como veremos en el Capítulo 3, la noción general del entorno como proveedor de un contexto en el que tiene lugar la evaluación jugará un papel importante en nuestra comprensión de la ejecución de programas.

Observe que la regla de evaluación dada anteriormente no maneja las definiciones. Por ejemplo, evaluar `(define x 3)` no aplica `define` a dos argumentos, uno de los cuales es el valor del símbolo `x` y el otro es 3, ya que el propósito de `define` es precisamente asociar `x` con un valor. (Es decir, `(define x 3)` no es una combinación).

Tales excepciones a la regla de evaluación general se llaman *formas especiales*. `define` es el único ejemplo de una forma especial que hemos visto hasta ahora, pero pronto encontraremos otras. Cada forma especial tiene su propia regla de evaluación. Los diversos tipos de expresiones (cada una con su regla de evaluación asociada) constituyen la sintaxis del lenguaje de programación. En comparación con la mayoría de los otros lenguajes de programación, Lisp tiene una sintaxis muy simple; es decir, la regla de evaluación para las expresiones puede describirse mediante una regla general simple junto con reglas especializadas para un pequeño número de formas especiales.[11]

Procedimientos compuestos

Hemos identificado en Lisp algunos de los elementos que deben aparecer en cualquier lenguaje de programación potente:

- Los números y las operaciones aritméticas son datos y procedimientos primitivos.

- El anidamiento de combinaciones proporciona un medio para combinar operaciones.

- Las definiciones que asocian nombres con valores proporcionan un medio limitado de abstracción.

Ahora aprenderemos sobre las *definiciones de procedimientos*, una técnica de abstracción mucho más potente mediante la cual una operación compuesta puede recibir un nombre y luego ser referida como una unidad.

Comenzamos examinando cómo expresar la idea de "elevar al cuadrado". Podríamos decir: "Para elevar algo al cuadrado, multiplícalo por sí mismo". Esto se expresa en nuestro lenguaje como

(define (square x) (* x x))

Podemos entender esto de la siguiente manera:

(define (square x) (* x x)) | | | | | | Para elevar al cuadrado algo, multiplícalo por sí mismo.

Tenemos aquí un *procedimiento compuesto*, al que se le ha dado el nombre `square`. El procedimiento representa la operación de multiplicar algo por sí mismo. A la cosa que se va a multiplicar se le da un nombre local, `x`, que desempeña el mismo papel que un pronombre en el lenguaje natural. La evaluación de la definición crea este procedimiento compuesto y lo asocia con el nombre `square`.[12]

La forma general de una definición de procedimiento es

(define (< *nombre* $\color{SchemeDark}\kern0.03em\rangle$ < *parámetros formales* $\color{SchemeDark}\kern0.02em\rangle$) $\color{SchemeDark}\langle\kern0.08em$ *cuerpo* >)

El $\langle\hbox{\sl nombre}\kern0.08em\rangle$ es un símbolo que se asociará con la definición del procedimiento en el entorno.[13] Los $\langle\hbox{\sl parámetros formales}\kern0.08em\rangle$ son los nombres utilizados dentro del cuerpo del procedimiento para referirse a los argumentos correspondientes del procedimiento. El $\langle\hbox{\sl cuerpo}\kern0.08em\rangle$ es una expresión que producirá el valor de la aplicación del procedimiento cuando los parámetros formales se reemplacen por los argumentos reales a los que se aplica el procedimiento.[14] El ⟨*nombre*$\kern0.08em\rangle$ y los ⟨*parámetros formales*$\kern0.08em\rangle$ se agrupan entre paréntesis, tal como lo estarían en una llamada real al procedimiento que se está definiendo.

Habiendo definido `square`, ahora podemos usarlo:

(square 21) *441* (square (+ 2 5)) *49* (square (square 3)) *81*

También podemos usar `square` como un bloque de construcción para definir

otros procedimientos. Por ejemplo, $x^2 + y^2$ se puede expresar como

(+ (square x) (square y))

Podemos definir fácilmente un procedimiento `sum/of/squares` que, dados dos números como argumentos, produce la suma de sus cuadrados:

(define (sum-of-squares x y) (+ (square x) (square y))) (sum-of-squares 3 4) *25*

Ahora podemos usar `sum/of/squares` como un bloque de construcción para construir más procedimientos:

(define (f a) (sum-of-squares (+ a 1) (* a 2))) (f 5) *136*

Los procedimientos compuestos se utilizan exactamente de la misma manera que los procedimientos primitivos. De hecho, al observar la definición de `sum/of/squares` dada anteriormente, no se podría decir si `square` estaba incorporado en el intérprete, como `+` y `*`, o si se definía como un procedimiento compuesto.

El modelo de sustitución para la aplicación de procedimientos

Para evaluar una combinación cuyo operador nombra a un procedimiento compuesto, el intérprete sigue un proceso muy similar al de las combinaciones cuyos operadores nombran procedimientos primitivos, que describimos en Section 1.1.3. Es decir, el intérprete evalúa los elementos de la combinación y aplica el procedimiento (que es el valor del operador de la combinación) a los argumentos (que son los valores de los operandos de la combinación).

Podemos suponer que el mecanismo para aplicar procedimientos primitivos a los argumentos está incorporado en el intérprete. Para los procedimientos compuestos, el proceso de aplicación es el siguiente:

> Para aplicar un procedimiento compuesto a los argumentos, evalúe el cuerpo del procedimiento con cada parámetro formal reemplazado por el argumento correspondiente.

Para ilustrar este proceso, evaluemos la combinación

(f 5)

donde `f` es el procedimiento definido en Section 1.1.4. Comenzamos recuperando el cuerpo de `f`:

(sum-of-squares (+ a 1) (* a 2))

Luego reemplazamos el parámetro formal `a` por el argumento 5:

(sum-of-squares (+ 5 1) (* 5 2))

Así, el problema se reduce a la evaluación de una combinación con dos operandos y un operador `sum/of/squares`. Evaluar esta combinación implica tres subproblemas. Debemos evaluar el operador para obtener el procedimiento a aplicar, y debemos evaluar los operandos para obtener los argumentos. Ahora `(+ 5 1)` produce 6 y `(* 5 2)` produce 10, por lo que debemos aplicar el procedimiento `sum/of/squares` a 6 y 10. Estos valores se sustituyen por los parámetros formales `x` e `y` en el cuerpo de `sum/of/squares`, reduciendo la expresión a

(+ (square 6) (square 10))

Si usamos la definición de `square`, esto se reduce a

(+ (* 6 6) (* 10 10))

que se reduce por multiplicación a

(+ 36 100)

y finalmente a

136

El proceso que acabamos de describir se llama el *modelo de sustitución* para la aplicación de procedimientos. Se puede tomar como un modelo que determina el "significado" de la aplicación de procedimientos, en lo que respecta a los procedimientos de este capítulo. Sin embargo, hay dos puntos que deben destacarse:

- El propósito de la sustitución es ayudarnos a pensar en la aplicación de procedimientos, no proporcionar una descripción de cómo funciona realmente el intérprete. Los intérpretes típicos no evalúan las aplicaciones de procedimientos manipulando el texto de un procedimiento para sustituir valores por los parámetros formales. En la práctica, la "sustitución" se logra utilizando un entorno local para los parámetros formales. Discutiremos esto más a fondo en el Capítulo 3 y en el Capítulo 4 cuando examinemos la implementación de un intérprete en detalle.

- A lo largo de este libro, presentaremos una secuencia de modelos cada vez más elaborados de cómo funcionan los intérpretes, que culminará con una implementación completa de un intérprete y un compilador en el Capítulo 5. El modelo de sustitución es solo el primero de estos modelos, una forma de comenzar a pensar formalmente en el proceso de evaluación. En general, al modelar fenómenos en ciencia e ingeniería, comenzamos con modelos simplificados e incompletos. A medida que examinamos las cosas con mayor detalle, estos modelos simples se vuelven inadecuados y deben ser reemplazados por modelos más refinados. El modelo de sustitución no es una excepción. En particular, cuando abordemos en el Capítulo 3 el uso de procedimientos con "datos mutables", veremos que el modelo de sustitución se rompe y debe ser reemplazado por un modelo más complicado de aplicación de procedimientos.[15]

Orden aplicativo frente a orden normal

Según la descripción de la evaluación dada en Section 1.1.3, el intérprete primero evalúa el operador y los operandos y luego aplica el procedimiento resultante a los argumentos resultantes. Esta no es la única forma de realizar la evaluación. Un modelo de evaluación alternativo no evaluaría los operandos hasta que se necesitaran sus valores. En su lugar, primero sustituiría las expresiones de los operandos por los parámetros hasta obtener una expresión que involucre solo operadores primitivos, y luego realizaría la evaluación. Si usáramos este método, la evaluación de `(f 5)` procedería según la secuencia de expansiones

(sum-of-squares (+ 5 1) (* 5 2)) (+ (square (+ 5 1)) (square (* 5 2))) (+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))

seguido de las reducciones

(+ (* 6 6) (* 10 10)) (+ 36 100) 136

Esto da la misma respuesta que nuestro modelo de evaluación anterior, pero el proceso es diferente. En particular, las evaluaciones de `(+ 5 1)` y `(* 5 2)` se realizan dos veces aquí, correspondiendo a la reducción de la expresión `(* x x)` con `x` reemplazado respectivamente por `(+ 5 1)` y `(* 5 2)`.

Este método de evaluación alternativo de "expandir completamente y luego reducir" se conoce como *evaluación en orden normal*, en contraste con el método de "evaluar los argumentos y luego aplicar" que el intérprete realmente usa, que se llama *evaluación en orden aplicativo*. Se puede demostrar que, para las aplicaciones de procedimientos que se pueden

modelar mediante sustitución (incluidos todos los procedimientos en los dos primeros capítulos de este libro) y que producen valores legítimos, la evaluación en orden normal y en orden aplicativo producen el mismo valor. (Consulte el Ejercicio 1.5 para un ejemplo de un valor "ilegítimo" donde la evaluación en orden normal y en orden aplicativo no dan el mismo resultado).

Lisp utiliza la evaluación en orden aplicativo, en parte debido a la eficiencia adicional obtenida al evitar múltiples evaluaciones de expresiones como las ilustradas con `(+ 5 1)` y `(* 5 2)` anteriormente y, de manera más significativa, porque la evaluación en orden normal se vuelve mucho más complicada de manejar cuando salimos del ámbito de los procedimientos que se pueden modelar por sustitución. Por otro lado, la evaluación en orden normal puede ser una herramienta extremadamente valiosa, e investigaremos algunas de sus implicaciones en el Capítulo 3 y en el Capítulo 4.[16]

Expresiones condicionales y predicados

El poder expresivo de la clase de procedimientos que podemos definir en este punto es muy limitado, porque no tenemos forma de hacer pruebas y realizar diferentes operaciones dependiendo del resultado de una prueba. Por ejemplo, no podemos definir un procedimiento que calcule el valor absoluto de un número probando si el número es positivo, negativo o cero y tomando diferentes acciones en los diferentes casos según la regla

$$|x| = \left\{ \begin{array}{r@{\quad \mathrm{si} \quad}l} x & x > 0, \\ \ 0 & x = 0, \\ \!\! -x & x < 0. \end{array} \right.$$

Esta construcción se llama *análisis de casos*, y hay una forma especial en Lisp para anotar tal análisis de casos. Se llama `cond` (que significa "condicional"), y se usa de la siguiente manera:

```
(define (abs x) (cond ((> x 0) x) ((= x 0) 0) ((< x 0) (- x))))
```

La forma general de una expresión condicional es

```
(cond ( < p $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$
 < e $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$ )
( < p $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$ < e $
\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$ ) ... ( < p $
\color{SchemeDark}_{\hbox{\ttfamily\itshape\scriptsize n}}\rangle$
 < e $\color{SchemeDark}_{\hbox{\ttfamily\itshape\scriptsize n}}\rangle
$ ))
```

que consiste en el símbolo `cond` seguido de pares de expresiones entre paréntesis

(⟨ *p* ⟩ ⟨ *e* ⟩)

llamadas *cláusulas*. La primera expresión en cada par es un *predicado*, es decir, una expresión cuyo valor se interpreta como verdadero o falso.[17]

Las expresiones condicionales se evalúan de la siguiente manera. El predicado $\langle p_1 \rangle$ se evalúa primero. Si su valor es falso, entonces se evalúa $\langle p_2 \rangle$. Si el valor de $\langle p_2 \rangle$ también es falso, entonces se evalúa $\langle p_3 \rangle$. Este proceso continúa hasta que se encuentra un predicado cuyo valor es verdadero, en cuyo caso el intérprete devuelve el valor de la *expresión consecuente* correspondiente $\langle e \rangle$ de la cláusula como el valor de la expresión condicional. Si no se encuentra que ninguno de los $\langle p \rangle$ sea verdadero, el valor de `cond` es indefinido.

La palabra *predicado* se usa para procedimientos que devuelven verdadero o falso, así como para expresiones que se evalúan como verdadero o falso. El procedimiento de valor absoluto `abs` hace uso de los predicados primitivos `>`, `<` y `=`.[18] Estos toman dos números como argumentos y prueban si el primer número es, respectivamente, mayor que, menor que o igual al segundo número, devolviendo verdadero o falso en consecuencia.

Otra forma de escribir el procedimiento de valor absoluto es

```
(define (abs x) (cond ((< x 0) (- x)) (else x)))
```

que podría expresarse en español como "Si x es menor que cero, devuelve $-x$; de lo contrario, devuelve x." `else` es un símbolo especial que se puede usar en lugar de $\langle p \rangle$ en la cláusula final de un `cond`. Esto hace que el `cond` devuelva como su valor el valor del $\langle e \rangle$ correspondiente cada vez que todas las cláusulas anteriores se hayan omitido. De hecho, cualquier expresión que siempre se evalúe como un valor verdadero podría usarse como el $\langle p \rangle$ aquí.

Aquí hay otra forma de escribir el procedimiento de valor absoluto:

```
(define (abs x) (if (< x 0) (- x) x))
```

Esto utiliza la forma especial `if`, un tipo restringido de condicional que se puede usar cuando hay precisamente dos casos en el análisis de casos. La forma general de una expresión `if` es

(if $\color{SchemeDark}\langle\kern0.07em$ *predicado* $\color{SchemeDark}\kern0.06em\rangle$ $\color{SchemeDark}\langle$

\kern0.07em$ *consecuente* $\color{SchemeDark}\kern0.05em\rangle$ $ \color{SchemeDark}\langle\kern0.06em$ *alternativa* $ \color{SchemeDark}\kern0.06em\rangle$)

Para evaluar una expresión `if`, el intérprete comienza evaluando la parte ⟨*predicado*$\kern0.04em\rangle$ de la expresión. Si el ⟨*predicado*$ \kern0.04em\rangle$ se evalúa como un valor verdadero, el intérprete luego evalúa el ⟨*consecuente*$\kern0.04em\rangle$ y devuelve su valor. De lo contrario, evalúa la ⟨*alternativa*$\kern0.04em\rangle$ y devuelve su valor.[19]

Además de los predicados primitivos como `<`, `=` y `>`, existen operaciones de composición lógica, que nos permiten construir predicados compuestos. Las tres más utilizadas son estas:

- $\hbox{\tt(and }\langle{e_1}\rangle\;\;\dots\;\;\langle{e_n}\rangle \hbox{\tt)}$

  El intérprete evalúa las expresiones $\langle{e}\kern0.08em\rangle$ una a la vez, en orden de izquierda a derecha. Si alguna $ \langle{e}\kern0.08em\rangle$ se evalúa como falso, el valor de la expresión `and` es falso, y el resto de las $\langle{e}\kern0.08em\rangle$ no se evalúan. Si todas las $\langle{e}\kern0.08em\rangle$ se evalúan como valores verdaderos, el valor de la expresión `and` es el valor de la última.

- $\hbox{\tt(or }\langle{e_1}\rangle\;\;\dots\;\;\langle{e_n}\rangle \hbox{\tt)}$

  El intérprete evalúa las expresiones $\langle{e}\kern0.08em\rangle$ una a la vez, en orden de izquierda a derecha. Si alguna $ \langle{e}\kern0.08em\rangle$ se evalúa como un valor verdadero, ese valor se devuelve como el valor de la expresión `or`, y el resto de las $ \langle{e}\kern0.08em\rangle$ no se evalúan. Si todas las $ \langle{e}\kern0.08em\rangle$ se evalúan como falso, el valor de la expresión `or` es falso.

- $\hbox{\tt(not }\langle{e}\rangle\hbox{\tt)}$

  El valor de una expresión `not` es verdadero cuando la expresión $ \langle{e}\kern0.08em\rangle$ se evalúa como falso, y falso en caso contrario.

Observe que `and` y `or` son formas especiales, no procedimientos, porque las subexpresiones no se evalúan necesariamente todas. `not` es un procedimiento ordinario.

Como ejemplo de cómo se usan, la condición de que un número x esté en el rango $5 < x < 10$ puede expresarse como

(and (> x 5) (< x 10))

Como otro ejemplo, podemos definir un predicado para probar si un número es mayor o igual que otro como

(define (>= x y) (or (> x y) (= x y)))

o alternativamente como

(define (>= x y) (not (< x y)))

Ejercicio 1.1: A continuación se presenta una secuencia de expresiones. ¿Cuál es el resultado impreso por el intérprete en respuesta a cada expresión? Suponga que la secuencia se evalúa en el orden en que se presenta.

10 (+ 5 3 4) (- 9 1) (/ 6 2) (+ (* 2 4) (- 4 6)) (define a 3) (define b (+ a 1)) (+ a b (* a b)) (= a b) (if (and (> b a) (< b (* a b))) b a)

(cond ((= a 4) 6) ((= b 4) (+ 6 7 a)) (else 25))

(+ 2 (if (> b a) b a))

(* (cond ((> a b) a) ((< a b) b) (else -1)) (+ a 1))

Ejercicio 1.2: Traduzca la siguiente expresión a forma prefija:

$${5 + 4 + (2 - (3 - (6 + {4\over5})))\over3(6 - 2)(2 - 7)}.$$

Ejercicio 1.3: Defina un procedimiento que tome tres números como argumentos y devuelva la suma de los cuadrados de los dos números más grandes.

Ejercicio 1.4: Observe que nuestro modelo de evaluación permite combinaciones cuyos operadores son expresiones compuestas. Use esta observación para describir el comportamiento del siguiente procedimiento:

(define (a-plus-abs-b a b) ((if (> b 0) + -) a b))

Ejercicio 1.5: Ben Bitdiddle ha inventado una prueba para determinar si el intérprete que está usando utiliza evaluación en orden aplicativo o evaluación en orden normal. Define los siguientes dos procedimientos:

(define (p) (p)) (define (test x y) (if (= x 0) 0 y))

Luego evalúa la expresión

(test 0 (p))

¿Qué comportamiento observará Ben con un intérprete que utiliza evaluación en orden aplicativo? ¿Qué comportamiento observará con un intérprete que utiliza evaluación en orden normal? Explique su respuesta. (Suponga que la regla de evaluación para la forma especial `if` es la misma ya sea que el intérprete esté usando orden normal o aplicativo: la expresión del predicado se evalúa primero, y el resultado determina si se evalúa la expresión consecuente o la alternativa).

Ejemplo: Raíces cuadradas por el método de Newton

Los procedimientos, como se introdujeron anteriormente, son muy parecidos a las funciones matemáticas ordinarias. Especifican un valor que está determinado por uno o más parámetros. Pero hay una diferencia importante entre las funciones matemáticas y los procedimientos informáticos. Los procedimientos deben ser efectivos.

Como ejemplo, consideremos el problema de calcular raíces cuadradas. Podemos definir la función de raíz cuadrada como

$$\sqrt{x}\;\; = {\rm\;\; la\;\;} y {\rm\;\; tal\;\; que\;\;} y \ge 0 {\rm\;\; y\;\;} y^2 = x.$$

Esto describe una función matemática perfectamente legítima. Podríamos usarla para reconocer si un número es la raíz cuadrada de otro, o para derivar hechos sobre las raíces cuadradas en general. Por otro lado, la definición no describe un procedimiento. De hecho, no nos dice casi nada sobre cómo encontrar realmente la raíz cuadrada de un número dado. No ayudará reformular esta definición en pseudo-Lisp:

(define (sqrt x) (the y (and (>= y 0) (= (square y) x))))

Esto solo evade la pregunta.

El contraste entre función y procedimiento es un reflejo de la distinción general entre describir propiedades de las cosas y describir cómo hacer las cosas, o, como a veces se le llama, la distinción entre conocimiento declarativo y conocimiento imperativo. En matemáticas, generalmente nos preocupamos por las descripciones declarativas (lo que es), mientras que en

la informática generalmente nos preocupamos por las descripciones imperativas (cómo hacer).[20]

¿Cómo se calculan las raíces cuadradas? La forma más común es usar el método de aproximaciones sucesivas de Newton, que dice que cada vez que tenemos una suposición *y* para el valor de la raíz cuadrada de un número *x*, podemos realizar una manipulación simple para obtener una mejor suposición (una más cercana a la raíz cuadrada real) promediando *y* con *x*/*y*.[21] Por ejemplo, podemos calcular la raíz cuadrada de 2 de la siguiente manera. Supongamos que nuestra suposición inicial es 1:

Suposición Cociente Promedio 1 (2/1) = 2 ((2 + 1)/2) = 1.5 1.5 (2/1.5) = 1.3333 ((1.3333 + 1.5)/2) = 1.4167 1.4167 (2/1.4167) = 1.4118 ((1.4167 + 1.4118)/2) = 1.4142 1.4142 ... ...

Continuando este proceso, obtenemos aproximaciones cada vez mejores a la raíz cuadrada.

Ahora formalicemos el proceso en términos de procedimientos. Comenzamos con un valor para el radicando (el número cuya raíz cuadrada estamos tratando de calcular) y un valor para la suposición. Si la suposición es lo suficientemente buena para nuestros propósitos, hemos terminado; si no, debemos repetir el proceso con una suposición mejorada. Escribimos esta estrategia básica como un procedimiento:

```
(define (sqrt-iter guess x) (if (good-enough? guess x) guess (sqrt-iter (improve guess x) x)))
```

Una suposición se mejora promediándola con el cociente del radicando y la suposición anterior:

```
(define (improve guess x) (average guess (/ x guess)))
```

donde

```
(define (average x y) (/ (+ x y) 2))
```

También tenemos que decir lo que queremos decir con "lo suficientemente bueno". Lo siguiente servirá como ilustración, pero en realidad no es una prueba muy buena. (Ver Ejercicio 1.7.) La idea es mejorar la respuesta hasta que esté lo suficientemente cerca como para que su cuadrado difiera del radicando en menos de una tolerancia predeterminada (aquí 0.001):[22]

```
(define (good-enough? guess x) (< (abs (- (square guess) x)) 0.001))
```

Finalmente, necesitamos una forma de empezar. Por ejemplo, siempre podemos suponer que la raíz cuadrada de cualquier número es 1:[23]

(define (sqrt x) (sqrt-iter 1.0 x))

Si escribimos estas definiciones en el intérprete, podemos usar `sqrt` tal como podemos usar cualquier procedimiento:

(sqrt 9) *3.00009155413138*

(sqrt (+ 100 37)) *11.704699917758145*

(sqrt (+ (sqrt 2) (sqrt 3))) *1.7739279023207892*

(square (sqrt 1000)) *1000.000369924366*

El programa `sqrt` también ilustra que el lenguaje procedimental simple que hemos introducido hasta ahora es suficiente para escribir cualquier programa puramente numérico que se podría escribir, digamos, en C o Pascal. Esto puede parecer sorprendente, ya que no hemos incluido en nuestro lenguaje ninguna construcción iterativa (de bucle) que ordene al ordenador hacer algo una y otra vez. `sqrt/iter`, por otro lado, demuestra cómo se puede lograr la iteración sin ninguna construcción especial que no sea la capacidad ordinaria de llamar a un procedimiento.[24]

Ejercicio 1.6: Alyssa P. Hacker no ve por qué `if` necesita ser proporcionado como una forma especial. "¿Por qué no puedo simplemente definirlo como un procedimiento ordinario en términos de `cond`?", pregunta. La amiga de Alyssa, Eva Lu Ator, afirma que esto sí se puede hacer, y define una nueva versión de `if`:

(define (new-if predicate then-clause else-clause) (cond (predicate then-clause) (else else-clause)))

Eva demuestra el programa a Alyssa:

(new-if (= 2 3) 0 5) *5* (new-if (= 1 1) 0 5) *0*

Encantada, Alyssa usa `new/if` para reescribir el programa de la raíz cuadrada:

(define (sqrt-iter guess x) (new-if (good-enough? guess x) guess (sqrt-iter (improve guess x) x)))

¿Qué sucede cuando Alyssa intenta usar esto para calcular raíces cuadradas? Explique.

Ejercicio 1.7: La prueba `good/enough?` utilizada para calcular raíces cuadradas no será muy efectiva para encontrar las raíces cuadradas de números muy pequeños. Además, en los ordenadores reales, las operaciones aritméticas casi siempre se realizan con precisión limitada. Esto hace que nuestra prueba sea inadecuada para números muy grandes. Explique estas afirmaciones, con ejemplos que muestren cómo la prueba falla para números pequeños y grandes. Una estrategia alternativa para implementar `good/enough?` es observar cómo cambia `guess` de una iteración a la siguiente y detenerse cuando el cambio es una fracción muy pequeña de la suposición. Diseñe un procedimiento de raíz cuadrada que utilice este tipo de prueba final. ¿Funciona mejor para números pequeños y grandes?

Ejercicio 1.8: El método de Newton para las raíces cúbicas se basa en el hecho de que si *y* es una aproximación a la raíz cúbica de *x*, entonces una mejor aproximación viene dada por el valor

$$${{x / y^2} + 2y \over 3}.$$

Use esta fórmula para implementar un procedimiento de raíz cúbica análogo al procedimiento de raíz cuadrada. (En Section 1.3.4 veremos cómo implementar el método de Newton en general como una abstracción de estos procedimientos de raíz cuadrada y raíz cúbica).

Procedimientos como abstracciones de caja negra

`sqrt` es nuestro primer ejemplo de un proceso definido por un conjunto de procedimientos mutuamente definidos. Observe que la definición de `sqrt/iter` es *recursiva*; es decir, el procedimiento se define en términos de sí mismo. La idea de poder definir un procedimiento en términos de sí mismo puede ser perturbadora; puede parecer poco claro cómo una definición tan "circular" podría tener sentido en absoluto, y mucho menos especificar un proceso bien definido para ser llevado a cabo por un ordenador. Esto se abordará con más cuidado en Section 1.2. Pero primero consideremos algunos otros puntos importantes ilustrados por el ejemplo de `sqrt`.

Observe que el problema de calcular raíces cuadradas se descompone naturalmente en una serie de subproblemas: cómo saber si una suposición es lo suficientemente buena, cómo mejorar una suposición, etc. Cada una de estas tareas se realiza mediante un procedimiento separado. El programa `sqrt` completo puede verse como un grupo de procedimientos (mostrado en la Figura 1.2) que refleja la descomposición del problema en subproblemas.

Figura 1.2: Descomposición procedimental del programa `sqrt`.

La importancia de esta estrategia de descomposición no es simplemente que se está dividiendo el programa en partes. Después de todo, podríamos tomar cualquier programa grande y dividirlo en partes: las primeras diez líneas, las siguientes diez líneas, las siguientes diez líneas, y así sucesivamente. Más bien, es crucial que cada procedimiento realice una tarea identificable que pueda usarse como un módulo en la definición de otros procedimientos. Por ejemplo, cuando definimos el procedimiento `good/enough?` en términos de `square`, podemos considerar el procedimiento `square` como una "caja negra". En ese momento no nos preocupa *cómo* el procedimiento calcula su resultado, solo el hecho de que calcula el cuadrado. Los detalles de cómo se calcula el cuadrado pueden suprimirse, para ser considerados en un momento posterior. De hecho, en lo que respecta al procedimiento `good/enough?`, `square` no es tanto un procedimiento como una abstracción de un procedimiento, una llamada *abstracción procedimental*. A este nivel de abstracción, cualquier procedimiento que calcule el cuadrado es igualmente bueno.

Por lo tanto, considerando solo los valores que devuelven, los siguientes dos procedimientos para elevar al cuadrado un número deberían ser indistinguibles. Cada uno toma un argumento numérico y produce el cuadrado de ese número como valor.[25]

(define (square x) (* x x)) (define (square x) (exp (double (log x)))) (define (double x) (+ x x))

Así, una definición de procedimiento debería poder suprimir detalles. Los usuarios del procedimiento pueden no haber escrito el procedimiento ellos mismos, sino que pueden haberlo obtenido de otro programador como una caja negra. Un usuario no debería necesitar saber cómo se implementa el procedimiento para poder usarlo.

Nombres locales

Un detalle de la implementación de un procedimiento que no debería importar al usuario del procedimiento es la elección del implementador de los nombres para los parámetros formales del procedimiento. Por lo tanto, los siguientes procedimientos no deberían ser distinguibles:

(define (square x) (* x x)) (define (square y) (* y y))

Este principio, que el significado de un procedimiento debe ser independiente de los nombres de los parámetros utilizados por su autor, parece a primera vista evidente, pero sus consecuencias son profundas. La consecuencia más

simple es que los nombres de los parámetros de un procedimiento deben ser locales al cuerpo del procedimiento. Por ejemplo, usamos `square` en la definición de `good/enough?` en nuestro procedimiento de raíz cuadrada:

```
(define (good-enough? guess x) (< (abs (- (square guess) x)) 0.001))
```

La intención del autor de `good/enough?` es determinar si el cuadrado del primer argumento está dentro de una tolerancia dada del segundo argumento. Vemos que el autor de `good/enough?` usó el nombre `guess` para referirse al primer argumento y `x` para referirse al segundo argumento. El argumento de `square` es `guess`. Si el autor de `square` usó `x` (como arriba) para referirse a ese argumento, vemos que el `x` en `good/enough?` debe ser un `x` diferente al de `square`. Ejecutar el procedimiento `square` no debe afectar el valor de `x` que es utilizado por `good/enough?`, porque ese valor de `x` puede ser necesario para `good/enough?` después de que `square` haya terminado de calcular.

Si los parámetros no fueran locales a los cuerpos de sus respectivos procedimientos, entonces el parámetro `x` en `square` podría confundirse con el parámetro `x` en `good/enough?`, y el comportamiento de `good/enough?` dependería de qué versión de `square` usáramos. Por lo tanto, `square` no sería la caja negra que deseábamos.

Un parámetro formal de un procedimiento tiene un papel muy especial en la definición del procedimiento, en el sentido de que no importa qué nombre tenga el parámetro formal. Tal nombre se llama una *variable ligada*, y decimos que la definición del procedimiento *liga* sus parámetros formales. El significado de una definición de procedimiento no cambia si una variable ligada se renombra consistentemente en toda la definición.[26] Si una variable no está ligada, decimos que es *libre*. El conjunto de expresiones para las cuales un enlace define un nombre se llama el *ámbito* de ese nombre. En una definición de procedimiento, las variables ligadas declaradas como los parámetros formales del procedimiento tienen el cuerpo del procedimiento como su ámbito.

En la definición de `good/enough?` anterior, `guess` y `x` son variables ligadas pero `<`, `-`, `abs` y `square` son libres. El significado de `good/enough?` debe ser independiente de los nombres que elijamos para `guess` y `x` siempre que sean distintos y diferentes de `<`, `-`, `abs` y `square`. (Si renombramos `guess` a `abs` habríamos introducido un error al *capturar* la variable `abs`. Habría pasado de libre a ligada.) Sin embargo, el significado de `good/enough?` no es independiente de los nombres de sus variables libres. Seguramente depende del hecho (externo a esta definición) de que el símbolo `abs` nombra un procedimiento para calcular el valor absoluto de un número. `good/enough?`

calculará una función diferente si sustituimos `cos` por `abs` en su definición.

Definiciones internas y estructura de bloques

Hasta ahora, tenemos un tipo de aislamiento de nombres disponible: los parámetros formales de un procedimiento son locales al cuerpo del procedimiento. El programa de la raíz cuadrada ilustra otra forma en la que nos gustaría controlar el uso de los nombres. El programa existente consiste en procedimientos separados:

(define (sqrt x) (sqrt-iter 1.0 x)) (define (sqrt-iter guess x) (if (good-enough? guess x) guess (sqrt-iter (improve guess x) x))) (define (good-enough? guess x) (< (abs (- (square guess) x)) 0.001)) (define (improve guess x) (average guess (/ x guess)))

El problema con este programa es que el único procedimiento que es importante para los usuarios de `sqrt` es `sqrt`. Los otros procedimientos (`sqrt/iter`, `good/enough?` e `improve`) solo desordenan sus mentes. No pueden definir ningún otro procedimiento llamado `good/enough?` como parte de otro programa para que funcione junto con el programa de la raíz cuadrada, porque `sqrt` lo necesita. El problema es especialmente grave en la construcción de grandes sistemas por muchos programadores separados. Por ejemplo, en la construcción de una gran biblioteca de procedimientos numéricos, muchas funciones numéricas se calculan como aproximaciones sucesivas y, por lo tanto, podrían tener procedimientos llamados `good/enough?` e `improve` como procedimientos auxiliares. Nos gustaría localizar los subprocedimientos, ocultándolos dentro de `sqrt` para que `sqrt` pueda coexistir con otras aproximaciones sucesivas, cada una con su propio procedimiento `good/enough?` privado. Para hacer esto posible, permitimos que un procedimiento tenga definiciones internas que son locales a ese procedimiento. Por ejemplo, en el problema de la raíz cuadrada podemos escribir

(define (sqrt x) (define (good-enough? guess x) (< (abs (- (square guess) x)) 0.001)) (define (improve guess x) (average guess (/ x guess))) (define (sqrt-iter guess x) (if (good-enough? guess x) guess (sqrt-iter (improve guess x) x))) (sqrt-iter 1.0 x))

Tal anidamiento de definiciones, llamado *estructura de bloques*, es básicamente la solución correcta al problema más simple de empaquetado de nombres. Pero aquí se esconde una idea mejor. Además de internalizar las definiciones de los procedimientos auxiliares, podemos simplificarlas. Dado que `x` está ligado en la definición de `sqrt`, los procedimientos `good/enough?`, `improve` y `sqrt/iter`, que se definen internamente a `sqrt`, están

en el ámbito de `x`. Por lo tanto, no es necesario pasar `x` explícitamente a cada uno de estos procedimientos. En su lugar, permitimos que `x` sea una variable libre en las definiciones internas, como se muestra a continuación. Entonces `x` obtiene su valor del argumento con el que se llama al procedimiento envolvente `sqrt`. Esta disciplina se llama *alcance léxico*.[27]

(define (sqrt x) (define (good-enough? guess) (< (abs (- (square guess) x)) 0.001)) (define (improve guess) (average guess (/ x guess))) (define (sqrt-iter guess) (if (good-enough? guess) guess (sqrt-iter (improve guess)))) (sqrt-iter 1.0))

Usaremos la estructura de bloques extensamente para ayudarnos a dividir grandes programas en piezas manejables.[28] La idea de la estructura de bloques se originó con el lenguaje de programación Algol 60. Aparece en la mayoría de los lenguajes de programación avanzados y es una herramienta importante para ayudar a organizar la construcción de grandes programas.

Procedimientos y los procesos que generan

Ya hemos considerado los elementos de la programación: hemos utilizado operaciones aritméticas primitivas, hemos combinado estas operaciones y hemos abstraído estas operaciones compuestas definiéndolas como procedimientos compuestos. Pero eso no es suficiente para que podamos decir que sabemos programar. Nuestra situación es análoga a la de alguien que ha aprendido las reglas de cómo se mueven las piezas en el ajedrez pero no sabe nada de aperturas, tácticas o estrategia típicas. Como el jugador de ajedrez novato, todavía no conocemos los patrones comunes de uso en el dominio. Nos falta el conocimiento de qué movimientos vale la pena hacer (qué procedimientos vale la pena definir). Nos falta la experiencia para predecir las consecuencias de hacer un movimiento (ejecutar un procedimiento).

La capacidad de visualizar las consecuencias de las acciones bajo consideración es crucial para convertirse en un programador experto, al igual que en cualquier actividad sintética y creativa. Al convertirse en un fotógrafo experto, por ejemplo, uno debe aprender a mirar una escena y saber qué tan oscura aparecerá cada región en una impresión para cada posible elección de exposición y condiciones de revelado. Solo entonces se puede razonar hacia atrás, planificando el encuadre, la iluminación, la exposición y el revelado para obtener los efectos deseados. Así es con la programación, donde estamos planeando el curso de acción que tomará un proceso y donde controlamos el proceso por medio de un programa. Para convertirnos en expertos, debemos aprender a visualizar los procesos generados por varios tipos de

procedimientos. Solo después de haber desarrollado tal habilidad podremos aprender a construir de manera confiable programas que exhiban el comportamiento deseado.

Un procedimiento es un patrón para la *evolución local* de un proceso computacional. Especifica cómo se construye cada etapa del proceso sobre la etapa anterior. Nos gustaría poder hacer afirmaciones sobre el comportamiento general, o *global*, de un proceso cuya evolución local ha sido especificada por un procedimiento. Esto es muy difícil de hacer en general, pero al menos podemos intentar describir algunos patrones típicos de la evolución de los procesos.

En esta sección examinaremos algunas "formas" comunes para los procesos generados por procedimientos simples. También investigaremos las tasas a las que estos procesos consumen los importantes recursos computacionales de tiempo y espacio. Los procedimientos que consideraremos son muy simples. Su papel es como el que desempeñan los patrones de prueba en la fotografía: como patrones prototípicos simplificados, en lugar de ejemplos prácticos por derecho propio.

Recursión lineal e iteración

Comenzamos considerando la función factorial, definida por

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdots 3 \cdot 2 \cdot 1.$$

Hay muchas maneras de calcular factoriales. Una forma es hacer uso de la observación de que $n!$ es igual a n por $(n - 1)!$ para cualquier entero positivo n:

$$n! = n \cdot [(n - 1) \cdot (n - 2) \cdots 3 \cdot 2 \cdot 1] = n \cdot (n - 1)!.$$

Por lo tanto, podemos calcular $n!$ calculando $(n - 1)!$ y multiplicando el resultado por n. Si añadimos la estipulación de que 1! es igual a 1, esta observación se traduce directamente en un procedimiento:

```
(define (factorial n) (if (= n 1) 1 (* n (factorial (- n 1)))))
```

Podemos usar el modelo de sustitución de Section 1.1.5 para ver este procedimiento en acción calculando 6!, como se muestra en la Figura 1.3.

Figura 1.3: Un proceso recursivo lineal para calcular 6!.

Ahora tomemos una perspectiva diferente sobre el cálculo de factoriales. Podríamos describir una regla para calcular *n*! especificando que primero multiplicamos 1 por 2, luego multiplicamos el resultado por 3, luego por 4, y así sucesivamente hasta que lleguemos a *n*. Más formalmente, mantenemos un producto acumulado, junto con un contador que cuenta desde 1 hasta *n*. Podemos describir el cálculo diciendo que el contador y el producto cambian simultáneamente de un paso al siguiente según la regla

producto ← contador * producto contador ← contador + 1

y estipulando que *n*! es el valor del producto cuando el contador excede *n*.

Figura 1.4: Un proceso iterativo lineal para calcular 6!.

Una vez más, podemos reformular nuestra descripción como un procedimiento para calcular factoriales:[29]

(define (factorial n) (fact-iter 1 1 n)) (define (fact-iter product counter max-count) (if (> counter max-count) product (fact-iter (* counter product) (+ counter 1) max-count)))

Como antes, podemos usar el modelo de sustitución para visualizar el proceso de calcular 6!, como se muestra en la Figura 1.4.

Compare los dos procesos. Desde un punto de vista, parecen apenas diferentes. Ambos calculan la misma función matemática en el mismo dominio, y cada uno requiere un número de pasos proporcional a *n* para calcular *n*!. De hecho, ambos procesos incluso llevan a cabo la misma secuencia de multiplicaciones, obteniendo la misma secuencia de productos parciales. Por otro lado, cuando consideramos las "formas" de los dos procesos, encontramos que evolucionan de manera bastante diferente.

Considere el primer proceso. El modelo de sustitución revela una forma de expansión seguida de contracción, indicada por la flecha en la Figura 1.3. La expansión ocurre a medida que el proceso construye una cadena de *operaciones diferidas* (en este caso, una cadena de multiplicaciones). La contracción ocurre a medida que las operaciones se realizan realmente. Este tipo de proceso, caracterizado por una cadena de operaciones diferidas, se llama un *proceso recursivo*. Llevar a cabo este proceso requiere que el intérprete haga un seguimiento de las operaciones que se realizarán más adelante. En el cálculo de *n*!, la longitud de la cadena de multiplicaciones diferidas, y por lo tanto la cantidad de información necesaria para hacer un seguimiento de ella, crece linealmente con *n* (es proporcional a *n*), al igual

que el número de pasos. Tal proceso se llama un *proceso recursivo lineal*.

Por el contrario, el segundo proceso no crece ni se encoge. En cada paso, todo lo que necesitamos para hacer un seguimiento, para cualquier *n*, son los valores actuales de las variables `product`, `counter` y `max/count`. A esto lo llamamos un *proceso iterativo*. En general, un proceso iterativo es uno cuyo estado puede resumirse mediante un número fijo de *variables de estado*, junto con una regla fija que describe cómo deben actualizarse las variables de estado a medida que el proceso se mueve de un estado a otro y una prueba final (opcional) que especifica las condiciones bajo las cuales el proceso debe terminar. Al calcular *n*!, el número de pasos requeridos crece linealmente con *n*. Tal proceso se llama un *proceso iterativo lineal*.

El contraste entre los dos procesos se puede ver de otra manera. En el caso iterativo, las variables del programa proporcionan una descripción completa del estado del proceso en cualquier punto. Si detuviéramos el cálculo entre pasos, todo lo que necesitaríamos hacer para reanudar el cálculo es proporcionar al intérprete los valores de las tres variables del programa. No es así con el proceso recursivo. En este caso, hay alguna información adicional "oculta", mantenida por el intérprete y no contenida en las variables del programa, que indica "dónde está el proceso" en la negociación de la cadena de operaciones diferidas. Cuanto más larga es la cadena, más información debe mantenerse.[30]

Al contrastar la iteración y la recursión, debemos tener cuidado de no confundir la noción de un *proceso* recursivo con la noción de un *procedimiento* recursivo. Cuando describimos un procedimiento como recursivo, nos referimos al hecho sintáctico de que la definición del procedimiento se refiere (directa o indirectamente) al procedimiento mismo. Pero cuando describimos un proceso como siguiendo un patrón que es, digamos, linealmente recursivo, estamos hablando de cómo evoluciona el proceso, no de la sintaxis de cómo se escribe un procedimiento. Puede parecer desconcertante que nos refiramos a un procedimiento recursivo como `fact/iter` como si generara un proceso iterativo. Sin embargo, el proceso es realmente iterativo: su estado se captura completamente por sus tres variables de estado, y un intérprete solo necesita hacer un seguimiento de tres variables para ejecutar el proceso.

Una razón por la que la distinción entre proceso y procedimiento puede ser confusa es que la mayoría de las implementaciones de lenguajes comunes (incluidos Ada, Pascal y C) están diseñadas de tal manera que la interpretación de cualquier procedimiento recursivo consume una cantidad de memoria que crece con el número de llamadas al procedimiento, incluso cuando el proceso descrito es, en principio, iterativo. Como consecuencia, estos lenguajes solo pueden describir procesos iterativos recurriendo a

construcciones de "bucle" de propósito especial como `do`, `repeat`, `until`, `for` y `while`. La implementación de Scheme que consideraremos en el Capítulo 5 no comparte este defecto. Ejecutará un proceso iterativo en un espacio constante, incluso si el proceso iterativo se describe mediante un procedimiento recursivo. Una implementación con esta propiedad se llama *recursiva de cola*. Con una implementación recursiva de cola, la iteración puede expresarse utilizando el mecanismo ordinario de llamada a procedimientos, de modo que las construcciones especiales de iteración son útiles solo como azúcar sintáctico.[31]

Ejercicio 1.9: Cada uno de los siguientes dos procedimientos define un método para sumar dos enteros positivos en términos de los procedimientos `inc`, que incrementa su argumento en 1, y `dec`, que decrementa su argumento en 1.

(define (+ a b) (if (= a 0) b (inc (+ (dec a) b)))) (define (+ a b) (if (= a 0) b (+ (dec a) (inc b))))

Usando el modelo de sustitución, ilustre el proceso generado por cada procedimiento al evaluar `(+ 4 5)`. ¿Son estos procesos iterativos o recursivos?

Ejercicio 1.10: El siguiente procedimiento calcula una función matemática llamada función de Ackermann.

(define (A x y) (cond ((= y 0) 0) ((= x 0) (* 2 y)) ((= y 1) 2) (else (A (- x 1) (A x (- y 1))))))

¿Cuáles son los valores de las siguientes expresiones?

(A 1 10) (A 2 4) (A 3 3)

Considere los siguientes procedimientos, donde `A` es el procedimiento definido anteriormente:

(define (f n) (A 0 n)) (define (g n) (A 1 n)) (define (h n) (A 2 n)) (define (k n) (* 5 n n))

Dé definiciones matemáticas concisas para las funciones calculadas por los procedimientos `f`, `g` y `h` para valores enteros positivos de *n*. Por ejemplo, `(k n)` calcula $5n^2$.

Recursión de árbol

Otro patrón común de computación se llama *recursión de árbol*. Como

ejemplo, considere el cálculo de la secuencia de números de Fibonacci, en la que cada número es la suma de los dos anteriores:

0, 1, 1, 2, 3, 5, 8, 13, 21, ….

En general, los números de Fibonacci se pueden definir por la regla

$${\rm Fib}(n) = \begin{cases} \; 0 & {\rm si} \;\; n=0, \\ \; 1 & {\rm si} \;\; n=1, \\ \; {\rm Fib}(n-1) + {\rm Fib}(n-2) \quad & {\rm en\;otro \;caso}. \end{cases}$$

Podemos traducir inmediatamente esta definición en un procedimiento recursivo para calcular los números de Fibonacci:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

Considere el patrón de este cálculo. Para calcular `(fib 5)`, calculamos `(fib 4)` y `(fib 3)`. Para calcular `(fib 4)`, calculamos `(fib 3)` y `(fib 2)`. En general, el proceso desarrollado se parece a un árbol, como se muestra en la Figura 1.5. Observe que las ramas se dividen en dos en cada nivel (excepto en la parte inferior); esto refleja el hecho de que el procedimiento `fib` se llama a sí mismo dos veces cada vez que se invoca.

Este procedimiento es instructivo como una recursión de árbol prototípica, pero es una forma terrible de calcular los números de Fibonacci porque realiza mucho cálculo redundante. Observe en la Figura 1.5 que todo el cálculo de `(fib 3)` —casi la mitad del trabajo— está duplicado. De hecho, no es difícil demostrar que el número de veces que el procedimiento calculará `(fib 1)` o `(fib 0)` (el número de hojas en el árbol anterior, en general) es precisamente Fib(n + 1). Para hacerse una idea de lo malo que es esto, se puede demostrar que el valor de Fib(n) crece exponencialmente con n. Más precisamente (ver Ejercicio 1.13), Fib(n) es el entero más cercano a $ \varphiˆn / \sqrt{5}$, donde

$$\varphi = {1 + \sqrt{5}\over2} \approx 1.6180$$

es la *razón áurea*, que satisface la ecuación

$$\varphi^2 = \varphi + 1.$$

Figura 1.5: El proceso de recursión de árbol generado al calcular `(fib 5)`.

Así, el proceso utiliza un número de pasos que crece exponencialmente con la entrada. Por otro lado, el espacio requerido crece solo linealmente con la entrada, porque solo necesitamos hacer un seguimiento de qué nodos están por encima de nosotros en el árbol en cualquier punto del cálculo. En general, el número de pasos requeridos por un proceso recursivo de árbol será proporcional al número de nodos en el árbol, mientras que el espacio requerido será proporcional a la profundidad máxima del árbol.

También podemos formular un proceso iterativo para calcular los números de Fibonacci. La idea es usar un par de enteros a y b, inicializados a Fib(1) $=$ 1 y Fib(0) $=$ 0, y aplicar repetidamente las transformaciones simultáneas

$$\begin{array}{l@{\;\;\gets\;\;}l} a & a + b, \\ b & a. \end{array}$$

No es difícil demostrar que, después de aplicar esta transformación n veces, a y b serán iguales, respectivamente, a Fib(n $+$ 1) y Fib(n). Así, podemos calcular los números de Fibonacci iterativamente usando el procedimiento

(define (fib n) (fib-iter 1 0 n)) (define (fib-iter a b count) (if (= count 0) b (fib-iter (+ a b) a (- count 1))))

Este segundo método para calcular Fib(n) es una iteración lineal. La diferencia en el número de pasos requeridos por los dos métodos —uno lineal en n, otro que crece tan rápido como el propio Fib(n)— es enorme, incluso para entradas pequeñas.

No se debe concluir de esto que los procesos recursivos de árbol son inútiles. Cuando consideremos procesos que operan sobre datos estructurados jerárquicamente en lugar de números, encontraremos que la recursión de árbol es una herramienta natural y poderosa.[32] Pero incluso en operaciones numéricas, los procesos recursivos de árbol pueden ser útiles para ayudarnos a comprender y diseñar programas. Por ejemplo, aunque el primer procedimiento `fib` es mucho menos eficiente que el segundo, es más directo, siendo poco más que una traducción a Lisp de la definición de la secuencia de Fibonacci. Para formular el algoritmo iterativo se requirió notar que el cálculo podía ser reformulado como una iteración con tres variables de estado.

Ejemplo: Contando el cambio

Solo se necesita un poco de ingenio para idear el algoritmo iterativo de Fibonacci. En contraste, considere el siguiente problema: ¿De cuántas maneras diferentes podemos dar cambio de $1.00, dados medios dólares, cuartos de dólar, monedas de diez centavos, monedas de cinco centavos y

centavos? Más generalmente, ¿podemos escribir un procedimiento para calcular el número de maneras de dar cambio para cualquier cantidad de dinero dada?

Este problema tiene una solución simple como un procedimiento recursivo. Supongamos que pensamos en los tipos de monedas disponibles como dispuestos en algún orden. Entonces se cumple la siguiente relación:

El número de maneras de dar cambio por la cantidad a usando n tipos de monedas es igual a

- el número de maneras de dar cambio por la cantidad a usando todos menos el primer tipo de moneda, más
- el número de maneras de dar cambio por la cantidad $a - d$ usando los n tipos de monedas, donde d es la denominación del primer tipo de moneda.

Para ver por qué esto es cierto, observe que las formas de dar cambio se pueden dividir en dos grupos: las que no usan ninguna del primer tipo de moneda y las que sí. Por lo tanto, el número total de formas de dar cambio para una cantidad es igual al número de formas de dar cambio para la cantidad sin usar ninguna del primer tipo de moneda, más el número de formas de dar cambio asumiendo que sí usamos el primer tipo de moneda. Pero este último número es igual al número de formas de dar cambio por la cantidad que queda después de usar una moneda del primer tipo.

Así, podemos reducir recursivamente el problema de dar cambio para una cantidad dada al problema de dar cambio para cantidades más pequeñas usando menos tipos de monedas. Considere esta regla de reducción cuidadosamente, y convénzase de que podemos usarla para describir un algoritmo si especificamos los siguientes casos degenerados:[33]

- Si a es exactamente 0, debemos contarlo como 1 forma de dar cambio.
- Si a es menor que 0, debemos contarlo como 0 formas de dar cambio.
- Si n es 0, debemos contarlo como 0 formas de dar cambio.

Podemos traducir fácilmente esta descripción en un procedimiento recursivo:

(define (count-change amount) (cc amount 5)) (define (cc amount kinds-of-coins) (cond ((= amount 0) 1) ((or (< amount 0) (= kinds-of-coins 0)) 0) (else (+ (cc amount (- kinds-of-coins 1)) (cc (- amount (first-denomination kinds-of-coins)) kinds-of-coins))))) (define (first-denomination kinds-of-coins)

```
(cond ((= kinds-of-coins 1) 1) ((= kinds-of-coins 2) 5) ((= kinds-of-coins 3)
10) ((= kinds-of-coins 4) 25) ((= kinds-of-coins 5) 50)))
```

(El procedimiento `first/denomination` toma como entrada el número de tipos de monedas disponibles y devuelve la denominación del primer tipo. Aquí estamos pensando en las monedas como dispuestas en orden de mayor a menor, pero cualquier orden funcionaría igual de bien.) Ahora podemos responder a nuestra pregunta original sobre dar cambio de un dólar:

```
(count-change 100)  292
```

`count/change` genera un proceso recursivo de árbol con redundancias similares a las de nuestra primera implementación de `fib`. (Tomará bastante tiempo para que se calcule ese 292.) Por otro lado, no es obvio cómo diseñar un algoritmo mejor para calcular el resultado, y dejamos este problema como un desafío. La observación de que un proceso recursivo de árbol puede ser muy ineficiente pero a menudo fácil de especificar y entender ha llevado a la gente a proponer que se podría obtener lo mejor de ambos mundos diseñando un "compilador inteligente" que pudiera transformar procedimientos recursivos de árbol en procedimientos más eficientes que calculen el mismo resultado.[34]

Ejercicio 1.11: Una función *f* está definida por la regla de que

$$f(n) = \begin{cases} \;\; n \quad \text{si \; \(n < 3 \),} \\ \;\; f(n-1) + 2\kern-0.08em f(n-2) + 3\kern-0.08em f(n-3) \quad \text{si \; \(n \ge 3 \).} \end{cases}$$

Escriba un procedimiento que calcule *f* por medio de un proceso recursivo. Escriba un procedimiento que calcule *f* por medio de un proceso iterativo.

Ejercicio 1.12: El siguiente patrón de números se llama *triángulo de Pascal*.

1 1 1 1 2 1 1 3 3 1 1 4 6 4 1 . . .

Los números en el borde del triángulo son todos 1, y cada número dentro del triángulo es la suma de los dos números que están sobre él.[35] Escriba un procedimiento que calcule los elementos del triángulo de Pascal por medio de un proceso recursivo.

Ejercicio 1.13: Demuestre que Fib(*n*) es el entero más cercano a $ \varphiˆn / \sqrt{5}$, donde $\varphi = (1 + \sqrt{5}) / 2$. Pista: Sea $\psi = (1 - \sqrt{5}) / 2$. Use inducción y la definición de los números

de Fibonacci (ver Section 1.2.2) para demostrar que $\text{Fib}(n) = (\varphiˆn - \psiˆn) / \sqrt{5}$.

Órdenes de crecimiento

Los ejemplos anteriores ilustran que los procesos pueden diferir considerablemente en las tasas a las que consumen recursos computacionales. Una forma conveniente de describir esta diferencia es usar la noción de *orden de crecimiento* para obtener una medida aproximada de los recursos requeridos por un proceso a medida que las entradas se hacen más grandes.

Sea n un parámetro que mide el tamaño del problema, y sea $R(n)$ la cantidad de recursos que el proceso requiere para un problema de tamaño n. En nuestros ejemplos anteriores tomamos n como el número para el cual se debe calcular una función dada, pero hay otras posibilidades. Por ejemplo, si nuestro objetivo es calcular una aproximación a la raíz cuadrada de un número, podríamos tomar n como el número de dígitos de precisión requeridos. Para la multiplicación de matrices, podríamos tomar n como el número de filas en las matrices. En general, hay una serie de propiedades del problema con respecto a las cuales será deseable analizar un proceso dado. De manera similar, $R(n)$ podría medir el número de registros de almacenamiento interno utilizados, el número de operaciones de máquina elementales realizadas, y así sucesivamente. En los ordenadores que solo realizan un número fijo de operaciones a la vez, el tiempo requerido será proporcional al número de operaciones de máquina elementales realizadas.

Decimos que $R(n)$ tiene un orden de crecimiento $\Theta(f(n))$, escrito $R(n) = \Theta(f(n))$ (pronunciado "theta de $f(n)$"), si existen constantes positivas k_1 y k_2 independientes de n tales que $k_1 f(n) \leq R(n) \leq k_2 f(n)$ para cualquier valor suficientemente grande de n. (En otras palabras, para n grande, el valor $R(n)$ está intercalado entre $k_1 f(n)$ y $k_2 f(n)$).

Por ejemplo, con el proceso recursivo lineal para calcular el factorial descrito en Section 1.2.1 el número de pasos crece proporcionalmente a la entrada n. Por lo tanto, los pasos requeridos para este proceso crecen como $\Theta(n)$. También vimos que el espacio requerido crece como $\Theta(n)$. Para el factorial iterativo, el número de pasos sigue siendo $\Theta(n)$ pero el espacio es $\Theta(1)$, es decir, constante.[36] El cálculo de Fibonacci recursivo de árbol requiere $\Theta(\varphi n)$ pasos y un espacio de $\Theta(n)$, donde φ es la razón áurea descrita en Section 1.2.2.

Los órdenes de crecimiento proporcionan solo una descripción aproximada del comportamiento de un proceso. Por ejemplo, un proceso que requiere n^2 pasos y un proceso que requiere $1000n^2$ pasos y un proceso que requiere $3n^2$

$+ 10n + 17$ pasos, todos tienen un orden de crecimiento de $\Theta(n^2)$. Por otro lado, el orden de crecimiento proporciona una indicación útil de cómo podemos esperar que cambie el comportamiento del proceso a medida que cambiamos el tamaño del problema. Para un proceso $\Theta(n)$ (lineal), duplicar el tamaño duplicará aproximadamente la cantidad de recursos utilizados. Para un proceso exponencial, cada incremento en el tamaño del problema multiplicará la utilización de recursos por un factor constante. En el resto de Section 1.2 examinaremos dos algoritmos cuyo orden de crecimiento es logarítmico, de modo que duplicar el tamaño del problema aumenta el requisito de recursos en una cantidad constante.

Ejercicio 1.14: Dibuje el árbol que ilustra el proceso generado por el procedimiento `count/change` de Section 1.2.2 al dar cambio de 11 centavos. ¿Cuáles son los órdenes de crecimiento del espacio y el número de pasos utilizados por este proceso a medida que aumenta la cantidad a cambiar?

Ejercicio 1.15: El seno de un ángulo (especificado en radianes) se puede calcular haciendo uso de la aproximación $\sin x \approx x$ si x es suficientemente pequeño, y la identidad trigonométrica

$$\sin x = 3\sin {x\over3} - 4\sin^3 {x\over3}$$

para reducir el tamaño del argumento de sin. (Para los propósitos de este ejercicio, un ángulo se considera "suficientemente pequeño" si su magnitud no es mayor que 0.1 radianes.) Estas ideas se incorporan en los siguientes procedimientos:

```
(define (cube x) (* x x x))
(define (p x) (- (* 3 x) (* 4 (cube x))))
(define (sine angle)
  (if (not (> (abs angle) 0.1))
      angle
      (p (sine (/ angle 3.0)))))
```

1. ¿Cuántas veces se aplica el procedimiento `p` cuando se evalúa `(sine 12.15)`?

2. ¿Cuál es el orden de crecimiento en espacio y número de pasos (como función de *a*) utilizado por el proceso generado por el procedimiento `sine` cuando se evalúa `(sine a)`?

Exponenciación

Considere el problema de calcular la exponencial de un número dado. Nos gustaría un procedimiento que tome como argumentos una base b y un exponente entero positivo n y calcule b^n. Una forma de hacerlo es a través de la definición recursiva

$$\begin{array}{l@{{}={}}l} b^n & b\cdot b^{n-1}, \\ b^0 & 1, \end{array}$$

que se traduce fácilmente en el procedimiento

```
(define (expt b n) (if (= n 0) 1 (* b (expt b (- n 1)))))
```

Este es un proceso recursivo lineal, que requiere $\Theta(n)$ pasos y $\Theta(n)$ de espacio. Al igual que con el factorial, podemos formular fácilmente una iteración lineal equivalente:

```
(define (expt b n) (expt-iter b n 1)) (define (expt-iter b counter product) (if (= counter 0) product (expt-iter b (- counter 1) (* b product))))
```

Esta versión requiere $\Theta(n)$ pasos y $\Theta(1)$ de espacio.

Podemos calcular exponenciales en menos pasos utilizando la elevación al cuadrado sucesiva. Por ejemplo, en lugar de calcular b^8 como

$$b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot b)))))),$$

podemos calcularlo usando tres multiplicaciones:

$$\begin{array}{l@{{}={}}l} b^2 & b\cdot b, \\ b^4 & b^2\cdot b^2, \\ b^8 & b^4\cdot b^4. \end{array}$$

Este método funciona bien para exponentes que son potencias de 2. También podemos aprovechar la elevación al cuadrado sucesiva para calcular exponenciales en general si usamos la regla

$$\begin{array}{l@{{}={}}lr@{\ n\ }l} b^n & (b^{n / 2})^2 \;\; & \mbox{si \,} & \mbox{\,es\, par}, \\ b^n & b\cdot b^{n-1} \;\; & \mbox{si\,} & \mbox{\,es\, impar}. \end{array}$$

Podemos expresar este método como un procedimiento:

```
(define (fast-expt b n) (cond ((= n 0) 1) ((even? n) (square (fast-expt b (/ n 2)))) (else (* b (fast-expt b (- n 1))))))
```

donde el predicado para probar si un entero es par se define en términos del procedimiento primitivo `remainder` por

```
(define (even? n) (= (remainder n 2) 0))
```

El proceso desarrollado por `fast/expt` crece logarítmicamente con n tanto en espacio como en número de pasos. Para ver esto, observe que calcular b^{2n}

usando `fast/expt` requiere solo una multiplicación más que calcular b^n. El tamaño del exponente que podemos calcular, por lo tanto, se duplica (aproximadamente) con cada nueva multiplicación que se nos permite. Por lo tanto, el número de multiplicaciones requeridas para un exponente de n crece aproximadamente tan rápido como el logaritmo de n en base 2. El proceso tiene un crecimiento de $\Theta(\log n)$.[37]

La diferencia entre el crecimiento $\Theta(\log n)$ y el crecimiento $\Theta(n)$ se vuelve sorprendente a medida que n se hace grande. Por ejemplo, `fast/expt` para $n = 1000$ requiere solo 14 multiplicaciones.[38] También es posible utilizar la idea de la elevación al cuadrado sucesiva para idear un algoritmo iterativo que calcule exponenciales con un número logarítmico de pasos (consulte el Ejercicio 1.16), aunque, como suele ser el caso con los algoritmos iterativos, esto no se escribe tan directamente como el algoritmo recursivo.[39]

Ejercicio 1.16: Diseñe un procedimiento que desarrolle un proceso de exponenciación iterativo que utilice la elevación al cuadrado sucesiva y utilice un número logarítmico de pasos, como lo hace `fast/expt`. (Pista: Usando la observación de que $(b^{n/2})^2 = (b^2)^{n/2}$, mantenga, junto con el exponente n y la base b, una variable de estado adicional a, y defina la transformación de estado de tal manera que el producto ab^n permanezca sin cambios de un estado a otro. Al principio del proceso, a se toma como 1, y la respuesta viene dada por el valor de a al final del proceso. En general, la técnica de definir una *cantidad invariante* que permanece sin cambios de un estado a otro es una forma poderosa de pensar en el diseño de algoritmos iterativos).

Ejercicio 1.17: Los algoritmos de exponenciación en esta sección se basan en realizar la exponenciación mediante la multiplicación repetida. De manera similar, se puede realizar la multiplicación de enteros mediante la suma repetida. El siguiente procedimiento de multiplicación (en el que se asume que nuestro lenguaje solo puede sumar, no multiplicar) es análogo al procedimiento `expt`:

```
(define (* a b) (if (= b 0) 0 (+ a (* a (- b 1)))))
```

Este algoritmo requiere un número de pasos que es lineal en `b`. Ahora suponga que incluimos, junto con la suma, las operaciones `double`, que duplica un entero, y `halve`, que divide un entero (par) por 2. Usando estas, diseñe un procedimiento de multiplicación análogo a `fast/expt` que utilice un número logarítmico de pasos.

Ejercicio 1.18: Usando los resultados de los Ejercicio 1.16 y Ejercicio 1.17, diseñe un procedimiento que genere un proceso iterativo para

multiplicar dos enteros en términos de sumar, duplicar y dividir por la mitad y que utilice un número logarítmico de pasos.[40]

Ejercicio 1.19: Hay un algoritmo ingenioso para calcular los números de Fibonacci en un número logarítmico de pasos. Recuerde la transformación de las variables de estado a y b en el proceso `fib/iter` de Section 1.2.2: $a \leftarrow a + b$ y $b \leftarrow a$. Llame a esta transformación T, y observe que aplicar T una y otra vez n veces, comenzando con 1 y 0, produce el par Fib($n + 1$) y Fib(n). En otras palabras, los números de Fibonacci se producen aplicando T^n, la n^{th} potencia de la transformación T, comenzando con el par (1, 0). Ahora considere T como el caso especial de $p = 0$ y $q = 1$ en una familia de transformaciones T_{pq}, donde T_{pq} transforma el par (a, b) según $a \leftarrow bq + aq + ap$ y $b \leftarrow bp + aq$. Demuestre que si aplicamos tal transformación T_{pq} dos veces, el efecto es el mismo que usar una sola transformación $T_{p'q'}$ de la misma forma, y calcule p' y q' en términos de p y q. Esto nos da una forma explícita de elevar al cuadrado estas transformaciones, y por lo tanto podemos calcular T^n usando la elevación al cuadrado sucesiva, como en el procedimiento `fast/expt`. Ponga todo esto junto para completar el siguiente procedimiento, que se ejecuta en un número logarítmico de pasos:[41]

(define (fib n) (fib-iter 1 0 0 1 n)) (define (fib-iter a b p q count) (cond ((= count 0) b) ((even? count) (fib-iter a b ⟨??⟩ ; calcular p' ⟨??⟩ ; calcular q' (/ count 2))) (else (fib-iter (+ (* b q) (* a q) (* a p)) (+ (* b p) (* a q)) p q (- count 1)))))

Máximo común divisor

El máximo común divisor (MCD) de dos enteros a y b se define como el entero más grande que divide tanto a a como a b sin dejar resto. Por ejemplo, el MCD de 16 y 28 es 4. En el Capítulo 2, cuando investiguemos cómo implementar la aritmética de números racionales, necesitaremos poder calcular MCDs para reducir los números racionales a su mínima expresión. (Para reducir un número racional a su mínima expresión, debemos dividir tanto el numerador como el denominador por su MCD. Por ejemplo, 16/28 se reduce a 4/7). Una forma de encontrar el MCD de dos enteros es factorizarlos y buscar factores comunes, pero hay un algoritmo famoso que es mucho más eficiente.

La idea del algoritmo se basa en la observación de que, si r es el resto cuando a se divide por b, entonces los divisores comunes de a y b son precisamente los mismos que los divisores comunes de b y r. Por lo tanto, podemos usar la

ecuación

MCD(a,b) = MCD(b,r)

para reducir sucesivamente el problema de calcular un MCD al problema de calcular el MCD de pares de enteros cada vez más pequeños. Por ejemplo,

MCD(206,40) = MCD(40,6) = MCD(6,4) = MCD(4,2) = MCD(2,0) = 2

reduce MCD(206, 40) a MCD(2, 0), que es 2. Es posible demostrar que comenzar con dos enteros positivos cualesquiera y realizar reducciones repetidas siempre producirá finalmente un par donde el segundo número es 0. Entonces, el MCD es el otro número en el par. Este método para calcular el MCD se conoce como el *Algoritmo de Euclides*.[42]

Es fácil expresar el Algoritmo de Euclides como un procedimiento:

(define (gcd a b) (if (= b 0) a (gcd b (remainder a b))))

Esto genera un proceso iterativo, cuyo número de pasos crece como el logaritmo de los números involucrados.

El hecho de que el número de pasos requeridos por el Algoritmo de Euclides tenga un crecimiento logarítmico tiene una relación interesante con los números de Fibonacci:

> **Teorema de Lamé:** Si el Algoritmo de Euclides requiere *k* pasos para calcular el MCD de algún par, entonces el número más pequeño del par debe ser mayor o igual que el k^{th} número de Fibonacci.[43]

Podemos usar este teorema para obtener una estimación del orden de crecimiento para el Algoritmo de Euclides. Sea *n* el menor de las dos entradas al procedimiento. Si el proceso toma *k* pasos, entonces debemos tener $n \ge {\rm Fib}(k) \approx \varphiˆk / \sqrt{5}$. Por lo tanto, el número de pasos *k* crece como el logaritmo (en base φ) de *n*. Por lo tanto, el orden de crecimiento es $\Theta(\log n)$.

> **Ejercicio 1.20:** El proceso que genera un procedimiento depende, por supuesto, de las reglas utilizadas por el intérprete. Como ejemplo, considere el procedimiento iterativo `gcd` dado anteriormente. Suponga que tuviéramos que interpretar este procedimiento utilizando la evaluación de orden normal, como se discutió en Section 1.1.5. (La regla de evaluación de orden normal para `if` se describe en el Ejercicio 1.5). Usando el método de sustitución (para orden normal), ilustre el proceso generado al evaluar `(gcd 206 40)` e indique las operaciones de `resto`

que se realizan realmente. ¿Cuántas operaciones de `resto` se realizan realmente en la evaluación de orden normal de `(gcd 206 40)`? ¿Y en la evaluación de orden aplicativo?

Ejemplo: Pruebas de primalidad

Esta sección describe dos métodos para verificar la primalidad de un entero *n*, uno con un orden de crecimiento de $\Theta(\sqrt{n})$, y un algoritmo "probabilístico" con un orden de crecimiento de $\Theta(\log n)$. Los ejercicios al final de esta sección sugieren proyectos de programación basados en estos algoritmos.

Búsqueda de divisores

Desde la antigüedad, los matemáticos han estado fascinados por los problemas relacionados con los números primos, y muchas personas han trabajado en el problema de determinar formas de probar si los números son primos. Una forma de probar si un número es primo es encontrar los divisores del número. El siguiente programa encuentra el divisor entero más pequeño (mayor que 1) de un número dado *n*. Lo hace de una manera sencilla, probando *n* para la divisibilidad por enteros sucesivos comenzando con 2.

(define (smallest-divisor n) (find-divisor n 2)) (define (find-divisor n test-divisor) (cond ((> (square test-divisor) n) n) ((divides? test-divisor n) test-divisor) (else (find-divisor n (+ test-divisor 1))))) (define (divides? a b) (= (remainder b a) 0))

Podemos probar si un número es primo de la siguiente manera: *n* es primo si y solo si *n* es su propio divisor más pequeño.

(define (prime? n) (= n (smallest-divisor n)))

La prueba final para `find/divisor` se basa en el hecho de que si *n* no es primo, debe tener un divisor menor o igual que $\sqrt{n}$.[44] Esto significa que el algoritmo solo necesita probar divisores entre 1 y $\sqrt{n}$. En consecuencia, el número de pasos requeridos para identificar *n* as prime will have order of growth $\Theta(\sqrt{n})$.

El test de Fermat

El test de primalidad $\Theta(\log n)$ se basa en un resultado de la teoría de números conocido como el Pequeño Teorema de Fermat.[45]

Pequeño Teorema de Fermat: Si n es un número primo y a es cualquier entero positivo menor que n, entonces a elevado a la n^{th} potencia es congruente con a módulo n.

(Se dice que dos números son *congruentes módulo n* si ambos tienen el mismo resto al ser divididos por n. El resto de un número a al ser dividido por n también se conoce como el *resto de a módulo n*, o simplemente como *a módulo n*.)

Si n no es primo, entonces, en general, la mayoría de los números $a < n$ no satisfarán la relación anterior. Esto conduce al siguiente algoritmo para probar la primalidad: dado un número n, elija un número aleatorio $a < n$ y calcule el resto de a^n módulo n. Si el resultado no es igual a a, entonces n ciertamente no es primo. Si es a, entonces hay buenas posibilidades de que n sea primo. Ahora elija otro número aleatorio a y pruébelo con el mismo método. Si también satisface la ecuación, entonces podemos estar aún más seguros de que n es primo. Al probar más y más valores de a, podemos aumentar nuestra confianza en el resultado. Este algoritmo se conoce como el test de Fermat.

Para implementar el test de Fermat, necesitamos un procedimiento que calcule la exponencial de un número módulo otro número:

(define (expmod base exp m) (cond ((= exp 0) 1) ((even? exp) (remainder (square (expmod base (/ exp 2) m)) m)) (else (remainder (* base (expmod base (- exp 1) m)) m))))

Esto es muy similar al procedimiento `fast/expt` de Section 1.2.4. Utiliza la exponenciación binaria, de modo que el número de pasos crece logarítmicamente con el exponente.[46]

El test de Fermat se realiza eligiendo al azar un número a entre 1 y $n - 1$ inclusive y verificando si el resto módulo n de la n^{th} potencia de a es igual a a. El número aleatorio a se elige usando el procedimiento `random`, que asumimos que está incluido como primitivo en Scheme. `random` devuelve un entero no negativo menor que su entrada entera. Por lo tanto, para obtener un número aleatorio entre 1 y $n - 1$, llamamos a `random` con una entrada de $n - 1$ y sumamos 1 al resultado:

(define (fermat-test n) (define (try-it a) (= (expmod a n n) a)) (try-it (+ 1 (random (- n 1)))))

El siguiente procedimiento ejecuta la prueba un número determinado de veces, según lo especificado por un parámetro. Su valor es verdadero si la prueba tiene éxito todas las veces, y falso en caso contrario.

(define (fast-prime? n times) (cond ((= times 0) true) ((fermat-test n) (fast-prime? n (- times 1))) (else false)))

Métodos probabilísticos

El test de Fermat difiere en su carácter de la mayoría de los algoritmos familiares, en los que se calcula una respuesta que se garantiza que es correcta. Aquí, la respuesta obtenida es solo probablemente correcta. Más precisamente, si *n* alguna vez falla el test de Fermat, podemos estar seguros de que *n* no es primo. Pero el hecho de que *n* pase la prueba, aunque es una indicación extremadamente fuerte, todavía no es una garantía de que *n* sea primo. Lo que nos gustaría decir es que para cualquier número *n*, si realizamos la prueba suficientes veces y encontramos que *n* siempre pasa la prueba, entonces la probabilidad de error en nuestra prueba de primalidad se puede hacer tan pequeña como queramos.

Desafortunadamente, esta afirmación no es del todo correcta. Existen números que engañan al test de Fermat: números *n* que no son primos y que sin embargo tienen la propiedad de que a^n es congruente con *a* módulo *n* para todos los enteros $a < n$. Dichos números son extremadamente raros, por lo que el test de Fermat es bastante fiable en la práctica.[47]

Existen variaciones del test de Fermat que no pueden ser engañadas. En estas pruebas, al igual que con el método de Fermat, se prueba la primalidad de un entero *n* eligiendo un entero aleatorio $a < n$ y verificando alguna condición que depende de *n* y *a*. (Ver Ejercicio 1.28 para un ejemplo de tal prueba.) Por otro lado, en contraste con el test de Fermat, se puede demostrar que, para cualquier *n*, la condición no se cumple para la mayoría de los enteros $a < n$ a menos que *n* sea primo. Por lo tanto, si *n* pasa la prueba para alguna elección aleatoria de *a*, las posibilidades son mayores que la mitad de que *n* sea primo. Si *n* pasa la prueba para dos elecciones aleatorias de *a*, las posibilidades son mayores que 3 de 4 de que *n* sea primo. Al ejecutar la prueba con más y más valores de *a* elegidos al azar, podemos hacer que la probabilidad de error sea tan pequeña como queramos.

La existencia de pruebas para las cuales se puede demostrar que la probabilidad de error se vuelve arbitrariamente pequeña ha despertado el interés en algoritmos de este tipo, que han llegado a ser conocidos como *algoritmos probabilísticos*. Hay una gran actividad de investigación en esta área, y los algoritmos probabilísticos se han aplicado fructíferamente en muchos campos.[48]

> **Ejercicio 1.21:** Use el procedimiento `smallest/divisor` para encontrar el divisor más pequeño de cada uno de los siguientes números: 199,

1999, 19999.

Ejercicio 1.22: La mayoría de las implementaciones de Lisp incluyen una primitiva llamada `runtime` que devuelve un entero que especifica la cantidad de tiempo que el sistema ha estado funcionando (medido, por ejemplo, en microsegundos). El siguiente procedimiento `timed/prime/test`, cuando se llama con un entero *n*, imprime *n* y comprueba si *n* es primo. Si *n* es primo, el procedimiento imprime tres asteriscos seguidos de la cantidad de tiempo utilizado para realizar la prueba.

(define (timed-prime-test n) (newline) (display n) (start-prime-test n (runtime))) (define (start-prime-test n start-time) (if (prime? n) (report-prime (- (runtime) start-time)))) (define (report-prime elapsed-time) (display " *** ") (display elapsed-time))

Usando este procedimiento, escriba un procedimiento `search/for/primes` que compruebe la primalidad de enteros impares consecutivos en un rango especificado. Use su procedimiento para encontrar los tres primos más pequeños mayores que 1000; mayores que 10,000; mayores que 100,000; mayores que 1,000,000. Anote el tiempo necesario para probar cada primo. Dado que el algoritmo de prueba tiene un orden de crecimiento de $\Theta(\sqrt{n})$, debería esperar que probar primos alrededor de 10,000 tome aproximadamente $\sqrt{10}$ veces más tiempo que probar primos alrededor de 1000. ¿Sus datos de tiempo confirman esto? ¿Qué tan bien los datos para 100,000 y 1,000,000 apoyan la predicción de $\Theta(\sqrt{n})$? ¿Es su resultado compatible con la noción de que los programas en su máquina se ejecutan en un tiempo proporcional al número de pasos requeridos para el cálculo?

Ejercicio 1.23: El procedimiento `smallest/divisor` que se muestra al principio de esta sección realiza muchas pruebas innecesarias: después de comprobar si el número es divisible por 2, no tiene sentido comprobar si es divisible por números pares más grandes. Esto sugiere que los valores utilizados para `test/divisor` no deberían ser 2, 3, 4, 5, 6, …, sino más bien 2, 3, 5, 7, 9, …. Para implementar este cambio, defina un procedimiento `next` que devuelva 3 si su entrada es igual a 2 y, de lo contrario, devuelva su entrada más 2. Modifique el procedimiento `smallest/divisor` para usar `(next test/divisor)` en lugar de `(+ test/divisor 1)`. Con `timed/prime/test` incorporando esta versión modificada de `smallest/divisor`, ejecute la prueba para cada uno de los 12 primos encontrados en el Ejercicio 1.22. Dado que esta modificación reduce a la mitad el número de pasos de prueba, debería esperar que se ejecute aproximadamente el doble de rápido. ¿Se confirma esta expectativa? Si no, ¿cuál es la relación observada de las velocidades de

los dos algoritmos y cómo explica el hecho de que sea diferente de 2?

Ejercicio 1.24: Modifique el procedimiento `timed/prime/test` del Ejercicio 1.22 para usar `fast/prime?` (el método de Fermat), y pruebe cada uno de los 12 primos que encontró en ese ejercicio. Dado que el test de Fermat tiene un crecimiento de $\Theta(\log n)$, ¿cómo esperaría que el tiempo para probar primos cerca de 1,000,000 se compare con el tiempo necesario para probar primos cerca de 1000? ¿Sus datos confirman esto? ¿Puede explicar cualquier discrepancia que encuentre?

Ejercicio 1.25: Alyssa P. Hacker se queja de que nos esforzamos mucho en escribir `expmod`. Después de todo, dice, como ya sabemos cómo calcular exponenciales, podríamos haber escrito simplemente

```
(define (expmod base exp m) (remainder (fast-expt base exp) m))
```

¿Tiene razón? ¿Serviría este procedimiento también para nuestro probador de primos rápido? Explique.

Ejercicio 1.26: Louis Reasoner tiene grandes dificultades para hacer el Ejercicio 1.24. Su prueba `fast/prime?` parece ejecutarse más lentamente que su prueba `prime?`. Louis llama a su amiga Eva Lu Ator para que lo ayude. Cuando examinan el código de Louis, encuentran que ha reescrito el procedimiento `expmod` para usar una multiplicación explícita, en lugar de llamar a `square`:

```
(define (expmod base exp m) (cond ((= exp 0) 1) ((even? exp)
(remainder (* (expmod base (/ exp 2) m) (expmod base (/ exp 2) m)) m))
(else (remainder (* base (expmod base (- exp 1) m)) m))))
```

"No veo qué diferencia podría hacer eso", dice Louis. "Yo sí", dice Eva. "Al escribir el procedimiento de esa manera, has transformado el proceso $\Theta(\log n)$ en un proceso $\Theta(n)$". Explique.

Ejercicio 1.27: Demuestre que los números de Carmichael enumerados en la Footnote 1.47 realmente engañan al test de Fermat. Es decir, escriba un procedimiento que tome un entero n y pruebe si a^n es congruente con a módulo n para cada $a < n$, y pruebe su procedimiento en los números de Carmichael dados.

Ejercicio 1.28: Una variante del test de Fermat que no puede ser engañada se llama el *test de Miller-Rabin* (Miller 1976; Rabin 1980). Este parte de una forma alternativa del Pequeño Teorema de Fermat, que establece que si n es un número primo y a es cualquier entero positivo menor que n, entonces a elevado a la potencia $(n - 1)$ es congruente con

1 módulo n. Para probar la primalidad de un número n mediante el test de Miller-Rabin, elegimos un número aleatorio $a < n$ y elevamos a a la potencia $(n - 1)$ módulo n usando el procedimiento `expmod`. Sin embargo, cada vez que realizamos el paso de elevar al cuadrado en `expmod`, comprobamos si hemos descubierto una "raíz cuadrada no trivial de 1 módulo n", es decir, un número no igual a 1 o $n - 1$ cuyo cuadrado es igual a 1 módulo n. Es posible demostrar que si existe tal raíz cuadrada no trivial de 1, entonces n no es primo. También es posible demostrar que si n es un número impar que no es primo, entonces, para al menos la mitad de los números $a < n$, calcular a^{n-1} de esta manera revelará una raíz cuadrada no trivial de 1 módulo n. (Esta es la razón por la que el test de Miller-Rabin no puede ser engañado). Modifique el procedimiento `expmod` para que indique si descubre una raíz cuadrada no trivial de 1, y use esto para implementar el test de Miller-Rabin con un procedimiento análogo a `fermat/test`. Verifique su procedimiento probando varios primos y no primos conocidos. Pista: una forma conveniente de hacer que `expmod` indique es que devuelva 0.

Formulando Abstracciones con Procedimientos de Orden Superior

Hemos visto que los procedimientos son, en efecto, abstracciones que describen operaciones compuestas sobre números independientemente de los números particulares. Por ejemplo, cuando

```
(define (cube x) (* x x x))
```

no estamos hablando del cubo de un número en particular, sino de un método para obtener el cubo de cualquier número. Por supuesto, podríamos prescindir de definir este procedimiento, escribiendo siempre expresiones como

```
(* 3 3 3) (* x x x) (* y y y)
```

y sin mencionar nunca `cube` explícitamente. Esto nos pondría en una seria desventaja, obligándonos a trabajar siempre al nivel de las operaciones particulares que resultan ser primitivas en el lenguaje (multiplicación, en este caso) en lugar de en términos de operaciones de nivel superior. Nuestros programas podrían calcular cubos, pero nuestro lenguaje carecería de la capacidad de expresar el concepto de elevar al cubo. Una de las cosas que deberíamos exigir de un lenguaje de programación potente es la capacidad de construir abstracciones asignando nombres a patrones comunes y luego

trabajar directamente en términos de las abstracciones. Los procedimientos proporcionan esta capacidad. Es por eso que todos los lenguajes de programación, excepto los más primitivos, incluyen mecanismos para definir procedimientos.

Sin embargo, incluso en el procesamiento numérico, estaremos severamente limitados en nuestra capacidad para crear abstracciones si nos restringimos a procedimientos cuyos parámetros deben ser números. A menudo, el mismo patrón de programación se usará con varios procedimientos diferentes. Para expresar tales patrones como conceptos, necesitaremos construir procedimientos que puedan aceptar procedimientos como argumentos o devolver procedimientos como valores. Los procedimientos que manipulan procedimientos se llaman *procedimientos de orden superior*. Esta sección muestra cómo los procedimientos de orden superior pueden servir como poderosos mecanismos de abstracción, aumentando enormemente el poder expresivo de nuestro lenguaje.

Procedimientos como Argumentos

Considere los siguientes tres procedimientos. El primero calcula la suma de los enteros desde `a` hasta `b`:

```
(define (sum-integers a b) (if (> a b) 0 (+ a (sum-integers (+ a 1) b))))
```

El segundo calcula la suma de los cubos de los enteros en el rango dado:

```
(define (sum-cubes a b) (if (> a b) 0 (+ (cube a) (sum-cubes (+ a 1) b))))
```

El tercero calcula la suma de una secuencia de términos de la serie

$${1\over1\cdot 3} + {1\over5\cdot 7} + {1\over9\cdot 11} + \dots,$$

que converge a $\pi/8$ (muy lentamente):[49]

```
(define (pi-sum a b) (if (> a b) 0 (+ (/ 1.0 (* a (+ a 2))) (pi-sum (+ a 4) b))))
```

Estos tres procedimientos comparten claramente un patrón subyacente común. Son en su mayor parte idénticos, difiriendo solo en el nombre del procedimiento, la función de `a` utilizada para calcular el término a sumar, y la función que proporciona el siguiente valor de `a`. Podríamos generar cada uno de los procedimientos rellenando espacios en la misma plantilla:

```
(define (⟨ nombre ⟩ a b) (if (> a b) 0 (+ (⟨ término ⟩ a) (⟨ nombre ⟩ (⟨ siguiente ⟩ a) b))))
```

La presencia de un patrón tan común es una fuerte evidencia de que hay una abstracción útil esperando a ser sacada a la superficie. De hecho, los matemáticos identificaron hace mucho tiempo la abstracción de la *sumatoria de una serie* e inventaron la "notación sigma", por ejemplo

$$\sum\limits_{n=a}^b f(n) = f(a) + \dots + f(b),$$

para expresar este concepto. El poder de la notación sigma es que permite a los matemáticos tratar con el concepto de sumatoria en sí mismo en lugar de solo con sumas particulares, por ejemplo, para formular resultados generales sobre sumas que son independientes de la serie particular que se está sumando.

Del mismo modo, como diseñadores de programas, nos gustaría que nuestro lenguaje fuera lo suficientemente potente como para que pudiéramos escribir un procedimiento que exprese el concepto de sumatoria en sí, en lugar de solo procedimientos que calculen sumas particulares. Podemos hacerlo fácilmente en nuestro lenguaje procedimental tomando la plantilla común que se muestra arriba y transformando los "espacios" en parámetros formales:

(define (sum term a next b) (if (> a b) 0 (+ (term a) (sum term (next a) next b))))

Observe que `sum` toma como argumentos los límites inferior y superior `a` y `b` junto con los procedimientos `term` y `next`. Podemos usar `sum` como lo haríamos con cualquier procedimiento. Por ejemplo, podemos usarlo (junto con un procedimiento `inc` que incrementa su argumento en 1) para definir `sum/cubes`:

(define (inc n) (+ n 1)) (define (sum-cubes a b) (sum cube a inc b))

Usando esto, podemos calcular la suma de los cubos de los enteros de 1 a 10:

(sum-cubes 1 10) *3025*

Con la ayuda de un procedimiento de identidad para calcular el término, podemos definir `sum/integers` en términos de `sum`:

(define (identity x) x) (define (sum-integers a b) (sum identity a inc b))

Luego podemos sumar los enteros de 1 a 10:

(sum-integers 1 10) *55*

También podemos definir `pi/sum` de la misma manera:[50]

(define (pi-sum a b) (define (pi-term x) (/ 1.0 (* x (+ x 2)))) (define (pi-next x) (+ x 4)) (sum pi-term a pi-next b))

Usando estos procedimientos, podemos calcular una aproximación a π:

(* 8 (pi-sum 1 1000)) *3.139592655589783*

Una vez que tenemos `sum`, podemos usarlo como un bloque de construcción para formular más conceptos. Por ejemplo, la integral definida de una función *f* entre los límites *a* y *b* puede aproximarse numéricamente usando la fórmula

$${\int_a^b \!\!\! f} = {\left[\;f\! \left(a + {dx \over 2}\right) + f\! \left(a + dx + {dx \over 2}\right) + f\! \left(a + 2dx + {dx \over 2}\right) + \, \dots \;\right]\! dx}$$

para valores pequeños de *dx*. Podemos expresar esto directamente como un procedimiento:

(define (integral f a b dx) (define (add-dx x) (+ x dx)) (* (sum f (+ a (/ dx 2.0)) add-dx b) dx))

(integral cube 0 1 0.01) *.24998750000000042*

(integral cube 0 1 0.001) *.249999875000001*

(El valor exacto de la integral de `cube` entre 0 y 1 es 1/4).

Ejercicio 1.29: La regla de Simpson es un método de integración numérica más preciso que el método ilustrado anteriormente. Usando la regla de Simpson, la integral de una función *f* entre *a* y *b* se aproxima como

$${h\over 3}(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \dots + 2y_{n-2} + 4y_{n-1} + y_n),$$

donde $h = (b - a)/n$, para algún entero par *n*, y $y_k = f(a + kh)$. (Aumentar *n* aumenta la precisión de la aproximación). Defina un procedimiento que tome como argumentos *f*, *a*, *b* y *n* y devuelva el valor de la integral, calculado usando la regla de Simpson. Use su procedimiento para integrar `cube` entre 0 y 1 (con $n = 100$ y $n = 1000$), y compare los resultados con los del procedimiento `integral` que se muestra arriba.

Ejercicio 1.30: El procedimiento `sum` anterior genera una recursión lineal. El procedimiento se puede reescribir para que la suma se realice

de forma iterativa. Muestre cómo hacerlo rellenando las expresiones que faltan en la siguiente definición:

(define (sum term a next b) (define (iter a result) (if < ?? > < ?? > (iter < ?? > < ?? >))) (iter < ?? > < ?? >))

Ejercicio 1.31:

1. El procedimiento `sum` es solo el más simple de una vasta cantidad de abstracciones similares que pueden capturarse como procedimientos de orden superior.[51] Escriba un procedimiento análogo llamado `product` que devuelva el producto de los valores de una función en puntos sobre un rango dado. Muestre cómo definir `factorial` en términos de `product`. También use `product` para calcular aproximaciones a π usando la fórmula[52]

   $${\pi\over 4} = {2\cdot 4\cdot 4\cdot 6\cdot 6\cdot 8\cdots\over 3\cdot 3\cdot 5\cdot 5\cdot 7\cdot 7\cdots}\,.$$

2. Si su procedimiento `product` genera un proceso recursivo, escriba uno que genere un proceso iterativo. Si genera un proceso iterativo, escriba uno que genere un proceso recursivo.

Ejercicio 1.32:

1. Muestre que `sum` y `product` (Ejercicio 1.31) son ambos casos especiales de una noción aún más general llamada `accumulate` que combina una colección de términos, usando alguna función de acumulación general:

   (accumulate combiner null-value term a next b)

   `accumulate` toma como argumentos las mismas especificaciones de término y rango que `sum` y `product`, junto con un procedimiento `combiner` (de dos argumentos) que especifica cómo se debe combinar el término actual con la acumulación de los términos precedentes y un `null/value` que especifica qué valor base usar cuando se agotan los términos. Escriba `accumulate` y muestre cómo `sum` y `product` pueden definirse ambos como llamadas simples a `accumulate`.

2. Si su procedimiento `accumulate` genera un proceso recursivo, escriba uno que genere un proceso iterativo. Si genera un proceso iterativo, escriba uno que genere un proceso recursivo.

Ejercicio 1.33: Puede obtener una versión aún más general de

`accumulate` (Ejercicio 1.32) introduciendo la noción de un *filtro* en los términos a combinar. Es decir, combine solo aquellos términos derivados de valores en el rango que satisfacen una condición especificada. La abstracción `filtered/accumulate` resultante toma los mismos argumentos que accumulate, junto con un predicado adicional de un argumento que especifica el filtro. Escriba `filtered/accumulate` como un procedimiento. Muestre cómo expresar lo siguiente usando `filtered/accumulate`:

1. la suma de los cuadrados de los números primos en el intervalo de *a* a *b* (suponiendo que ya tiene un predicado `prime?` escrito)

2. el producto de todos los enteros positivos menores que *n* que son coprimos con *n* (es decir, todos los enteros positivos $i < n$ tales que $ \textsc{mcd}(i, n) = 1$).

Construyendo Procedimientos Usando `lambda`

Al usar `sum` como en Section 1.3.1, parece terriblemente incómodo tener que definir procedimientos triviales como `pi/term` y `pi/next` solo para poder usarlos como argumentos de nuestro procedimiento de orden superior. En lugar de definir `pi/next` y `pi/term`, sería más conveniente tener una forma de especificar directamente "el procedimiento que devuelve su entrada incrementada en 4" y "el procedimiento que devuelve el recíproco de su entrada por su entrada más 2". Podemos hacer esto introduciendo la forma especial `lambda`, que crea procedimientos. Usando `lambda` podemos describir lo que queremos como

(lambda (x) (+ x 4))

y

(lambda (x) (/ 1.0 (* x (+ x 2))))

Entonces nuestro procedimiento `pi/sum` se puede expresar sin definir ningún procedimiento auxiliar como

(define (pi-sum a b) (sum (lambda (x) (/ 1.0 (* x (+ x 2)))) a (lambda (x) (+ x 4)) b))

Usando de nuevo `lambda`, podemos escribir el procedimiento `integral` sin tener que definir el procedimiento auxiliar `add/dx`:

(define (integral f a b dx) (* (sum f (+ a (/ dx 2.0)) (lambda (x) (+ x dx)) b)

dx))

En general, `lambda` se usa para crear procedimientos de la misma manera que `define`, excepto que no se especifica ningún nombre para el procedimiento:

(lambda (< *parámetros-formales* >) < *cuerpo* >)

El procedimiento resultante es tan procedimiento como uno que se crea usando `define`. La única diferencia es que no se ha asociado con ningún nombre en el entorno. De hecho,

(define (plus4 x) (+ x 4))

es equivalente a

(define plus4 (lambda (x) (+ x 4)))

Podemos leer una expresión `lambda` de la siguiente manera:

(lambda (x) (+ x 4)) | | | | | el procedimiento de un argumento x que suma x y 4

Como cualquier expresión que tiene un procedimiento como valor, una expresión `lambda` puede usarse como el operador en una combinación como

((lambda (x y z) (+ x y (square z))) 1 2 3) *12*

o, más generalmente, en cualquier contexto donde normalmente usaríamos un nombre de procedimiento.[53]

Usando `let` para crear variables locales

Otro uso de `lambda` es en la creación de variables locales. A menudo necesitamos variables locales en nuestros procedimientos distintas de las que se han ligado como parámetros formales. Por ejemplo, supongamos que deseamos calcular la función

$$f(x,y) = x(1 + xy)^2 + y(1 - y) + (1 + xy)(1 - y),$$

que también podríamos expresar como

$$\begin{array}{r@{{}={}}l} a & 1 + xy, \\ b & 1 - y, \\ f(x,y) & xa^2 + yb + ab. \end{array}$$

Al escribir un procedimiento para calcular *f*, nos gustaría incluir como

variables locales no solo *x* y *y*, sino también los nombres de cantidades intermedias como *a* y *b*. Una forma de lograr esto es usar un procedimiento auxiliar para ligar las variables locales:

(define (f x y) (define (f-helper a b) (+ (* x (square a)) (* y b) (* a b))) (f-helper (+ 1 (* x y)) (- 1 y)))

Por supuesto, podríamos usar una expresión `lambda` para especificar un procedimiento anónimo para ligar nuestras variables locales. El cuerpo de `f` se convierte entonces en una sola llamada a ese procedimiento:

(define (f x y) ((lambda (a b) (+ (* x (square a)) (* y b) (* a b))) (+ 1 (* x y)) (- 1 y)))

Esta construcción es tan útil que hay una forma especial llamada `let` para hacer su uso más conveniente. Usando `let`, el procedimiento `f` podría escribirse como

(define (f x y) (let ((a (+ 1 (* x y))) (b (- 1 y))) (+ (* x (square a)) (* y b) (* a b))))

La forma general de una expresión `let` es

(let ((< *var* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$ < *exp* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$) (< *var* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$ < *exp* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$) … (< *var* $\color{SchemeDark}_{\hbox{\ttfamily\itshape\scriptsize n}}\rangle$ < *exp* $\color{SchemeDark}_{\hbox{\ttfamily\itshape\scriptsize n}}\rangle$)) < *body* >)

que se puede pensar como si dijera

sea < *var* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$ con el valor < *exp* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$ y < *var* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$ con el valor < *exp* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$ y … < *var* $\color{SchemeDark}_{\hbox{\ttfamily\itshape\scriptsize n}}\rangle$ con el valor < *exp* $\color{SchemeDark}_{\hbox{\ttfamily\itshape\scriptsize n}}\rangle$ en < *body* >

La primera parte de la expresión `let` es una lista de pares nombre-expresión. Cuando se evalúa el `let`, cada nombre se asocia con el valor de la expresión correspondiente. El cuerpo del `let` se evalúa con estos nombres ligados como

variables locales. La forma en que esto sucede es que la expresión `let` se interpreta como una sintaxis alternativa para

((lambda (< *var* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$ … < *var* $\color{SchemeDark}_{\hbox{\ttfamily\itshape \scriptsize n}}\rangle$) < *body* >) < *exp* $\color{SchemeDark} _{\hbox{\ttfamily\scriptsize 1}}\rangle$ … < *exp* $\color{SchemeDark} _{\hbox{\ttfamily\itshape\scriptsize n}}\rangle$)

No se requiere ningún mecanismo nuevo en el intérprete para proporcionar variables locales. Una expresión `let` es simplemente azúcar sintáctico para la aplicación `lambda` subyacente.

Podemos ver por esta equivalencia que el ámbito de una variable especificada por una expresión `let` es el cuerpo del `let`. Esto implica que:

- `let` permite ligar variables lo más localmente posible a donde se van a usar. Por ejemplo, si el valor de `x` es 5, el valor de la expresión

  (+ (let ((x 3)) (+ x (* x 10))) x)

  es 38. Aquí, la `x` en el cuerpo del `let` es 3, por lo que el valor de la expresión `let` es 33. Por otro lado, la `x` que es el segundo argumento del `+` más externo sigue siendo 5.

- Los valores de las variables se calculan fuera del `let`. Esto importa cuando las expresiones que proporcionan los valores para las variables locales dependen de variables que tienen los mismos nombres que las propias variables locales. Por ejemplo, si el valor de `x` es 2, la expresión

  (let ((x 3) (y (+ x 2))) (* x y))

  tendrá el valor 12 porque, dentro del cuerpo del `let`, `x` será 3 e `y` será 4 (que es la `x` externa más 2).

A veces podemos usar definiciones internas para obtener el mismo efecto que con `let`. Por ejemplo, podríamos haber definido el procedimiento `f` anterior como

(define (f x y) (define a (+ 1 (* x y))) (define b (- 1 y)) (+ (* x (square a)) (* y b) (* a b)))

Preferimos, sin embargo, usar `let` en situaciones como esta y usar `define` interno solo para procedimientos internos.[54]

Ejercicio 1.34: Supongamos que definimos el procedimiento

(define (f g) (g 2))

Entonces tenemos

(f square) *4* (f (lambda (z) (* z (+ z 1)))) *6*

¿Qué sucede si (perversamente) le pedimos al intérprete que evalúe la combinación `(f f)`? Explique.

Procedimientos como Métodos Generales

Introdujimos los procedimientos compuestos en Section 1.1.4 como un mecanismo para abstraer patrones de operaciones numéricas para independizarlas de los números particulares involucrados. Con procedimientos de orden superior, como el procedimiento `integral` de Section 1.3.1, comenzamos a ver un tipo de abstracción más potente: procedimientos utilizados para expresar métodos generales de computación, independientes de las funciones particulares involucradas. En esta sección discutimos dos ejemplos más elaborados —métodos generales para encontrar ceros y puntos fijos de funciones— y mostramos cómo estos métodos pueden expresarse directamente como procedimientos.

Encontrando raíces de ecuaciones por el método de la bisección

El *método de la bisección* es una técnica simple pero potente para encontrar raíces de una ecuación $f(x) = 0$, donde f es una función continua. La idea es que, si se nos dan puntos a y b tales que $f(a) < 0 < f(b)$, entonces f debe tener al menos un cero entre a y b. Para localizar un cero, sea x el promedio de a y b, y calcule $f(x)$. Si $f(x) > 0$, entonces f debe tener un cero entre a y x. Si $f(x) < 0$, entonces f debe tener un cero entre x y b. Continuando de esta manera, podemos identificar intervalos cada vez más pequeños en los que f debe tener un cero. Cuando llegamos a un punto donde el intervalo es lo suficientemente pequeño, el proceso se detiene. Dado que el intervalo de incertidumbre se reduce a la mitad en cada paso del proceso, el número de pasos requeridos crece como $\Theta(\log(L/T))$, donde L es la longitud del intervalo original y T es la tolerancia de error (es decir, el tamaño del intervalo que consideraremos "suficientemente pequeño"). Aquí hay un procedimiento que implementa esta estrategia:

(define (search f neg-point pos-point) (let ((midpoint (average neg-point pos-point))) (if (close-enough? neg-point pos-point) midpoint (let ((test-value (f midpoint))) (cond ((positive? test-value) (search f neg-point midpoint))

((negative? test-value) (search f midpoint pos-point)) (else midpoint))))))

Suponemos que inicialmente se nos da la función *f* junto con puntos en los que sus valores son negativos y positivos. Primero calculamos el punto medio de los dos puntos dados. Luego verificamos si el intervalo dado es lo suficientemente pequeño, y si es así, simplemente devolvemos el punto medio como nuestra respuesta. De lo contrario, calculamos como valor de prueba el valor de *f* en el punto medio. Si el valor de prueba es positivo, entonces continuamos el proceso con un nuevo intervalo que va desde el punto negativo original hasta el punto medio. Si el valor de prueba es negativo, continuamos con el intervalo desde el punto medio hasta el punto positivo. Finalmente, existe la posibilidad de que el valor de prueba sea 0, en cuyo caso el punto medio es en sí mismo la raíz que estamos buscando.

Para probar si los puntos finales están "suficientemente cerca", podemos usar un procedimiento similar al utilizado en Section 1.1.7 para calcular raíces cuadradas:[55]

(define (close-enough? x y) (< (abs (- x y)) 0.001))

`search` es incómodo de usar directamente, porque accidentalmente podemos darle puntos en los que los valores de *f* no tienen el signo requerido, en cuyo caso obtenemos una respuesta incorrecta. En su lugar, usaremos `search` a través del siguiente procedimiento, que comprueba cuál de los puntos finales tiene un valor de función negativo y cuál tiene un valor positivo, y llama al procedimiento `search` en consecuencia. Si la función tiene el mismo signo en los dos puntos dados, no se puede usar el método de bisección, en cuyo caso el procedimiento señala un error.[56]

(define (half-interval-method f a b) (let ((a-value (f a)) (b-value (f b))) (cond ((and (negative? a-value) (positive? b-value)) (search f a b)) ((and (negative? b-value) (positive? a-value)) (search f b a)) (else (error "Values are not of opposite sign" a b)))))

El siguiente ejemplo utiliza el método de la bisección para aproximar π como la raíz entre 2 y 4 de $\sin x = 0$:

(half-interval-method sin 2.0 4.0) *3.14111328125*

Aquí hay otro ejemplo, usando el método de la bisección para buscar una raíz de la ecuación $x^3 - 2x - 3 = 0$ entre 1 y 2:

(half-interval-method (lambda (x) (- (* x x x) (* 2 x) 3)) 1.0 2.0)
1.89306640625

Encontrando puntos fijos de funciones

Un número x se llama un *punto fijo* de una función f si x satisface la ecuación $f(x) = x$. Para algunas funciones f podemos localizar un punto fijo comenzando con una suposición inicial y aplicando f repetidamente,

$$f(x), \quad f(f(x)), \quad f(f(f(x))), \quad \ldots,$$

hasta que el valor no cambie mucho. Usando esta idea, podemos idear un procedimiento `fixed/point` que tome como entradas una función y una suposición inicial y produzca una aproximación a un punto fijo de la función. Aplicamos la función repetidamente hasta que encontramos dos valores sucesivos cuya diferencia es menor que alguna tolerancia prescrita:

(define tolerance 0.00001) (define (fixed-point f first-guess) (define (close-enough? v1 v2) (< (abs (- v1 v2)) tolerance)) (define (try guess) (let ((next (f guess))) (if (close-enough? guess next) next (try next)))) (try first-guess))

Por ejemplo, podemos usar este método para aproximar el punto fijo de la función coseno, comenzando con 1 como aproximación inicial:[57]

(fixed-point cos 1.0) *.7390822985224023*

Del mismo modo, podemos encontrar una solución a la ecuación $y = \sin y + \cos y$:

(fixed-point (lambda (y) (+ (sin y) (cos y))) 1.0) *1.2587315962971173*

El proceso de punto fijo recuerda al proceso que usamos para encontrar raíces cuadradas en Section 1.1.7. Ambos se basan en la idea de mejorar repetidamente una suposición hasta que el resultado satisfaga algún criterio. De hecho, podemos formular fácilmente el cálculo de la raíz cuadrada como una búsqueda de punto fijo. Calcular la raíz cuadrada de algún número x requiere encontrar un y tal que $y^2 = x$. Poniendo esta ecuación en la forma equivalente $y = x/y$, reconocemos que estamos buscando un punto fijo de la función[58] $y \mapsto x/y$, y por lo tanto podemos intentar calcular las raíces cuadradas como

(define (sqrt x) (fixed-point (lambda (y) (/ x y)) 1.0))

Desafortunadamente, esta búsqueda de punto fijo no converge. Considere una suposición inicial y_1. La siguiente suposición es $y_2 = x/y_1$ y la siguiente suposición es $y_3 = x/y_2 = x/(x/y_1) = y_1$. Esto resulta en un bucle infinito en el que las dos suposiciones y_1 y y_2 se repiten una y otra vez, oscilando alrededor de la respuesta.

Una forma de controlar tales oscilaciones es evitar que las suposiciones cambien tanto. Dado que la respuesta siempre está entre nuestra suposición *y* y *x/y*, podemos hacer una nueva suposición que no esté tan lejos de *y* como *x/y* promediando *y* con *x/y*, de modo que la siguiente suposición después de *y* sea ${1\over2}(y + x / y)$ en lugar de *x/y*. El proceso de hacer tal secuencia de suposiciones es simplemente el proceso de buscar un punto fijo de $y \mapsto {1\over2}(y + x / y)$:

```
(define (sqrt x) (fixed-point (lambda (y) (average y (/ x y))) 1.0))
```

(Tenga en cuenta que $y = {1\over2}(y + x / y)$ es una simple transformación de la ecuación $y = x/y$; para derivarla, sume *y* a ambos lados de la ecuación y divida por 2).

Con esta modificación, el procedimiento de la raíz cuadrada funciona. De hecho, si desentrañamos las definiciones, podemos ver que la secuencia de aproximaciones a la raíz cuadrada generada aquí es precisamente la misma que la generada por nuestro procedimiento original de raíz cuadrada de Section 1.1.7. Este enfoque de promediar aproximaciones sucesivas a una solución, una técnica que llamamos *amortiguación promedio*, a menudo ayuda a la convergencia de las búsquedas de punto fijo.

Ejercicio 1.35: Muestre que la razón áurea φ (Section 1.2.2) es un punto fijo de la transformación $x \mapsto 1 + 1/x$, y use este hecho para calcular φ por medio del procedimiento `fixed/point`.

Ejercicio 1.36: Modifique `fixed/point` para que imprima la secuencia de aproximaciones que genera, usando las primitivas `newline` y `display` que se muestran en el Ejercicio 1.22. Luego encuentre una solución a $x^x = 1000$ encontrando un punto fijo de $x \mapsto \log(1000)/\log(x)$. (Use el procedimiento primitivo de Scheme `log`, que calcula logaritmos naturales). Compare el número de pasos que esto toma con y sin amortiguación promedio. (Tenga en cuenta que no puede iniciar `fixed/point` con una suposición de 1, ya que esto causaría una división por $\log(1) = 0$).

Ejercicio 1.37:

1. Una *fracción continua* infinita es una expresión de la forma

$$${f} = \cfrac{N_1}{D_1 + \cfrac{N_2}{D_2 + \cfrac{N_3}{D_3 + \dots}}}\,.$$

   Como ejemplo, se puede demostrar que la expansión de fracción continua infinita con todos los N_i y D_i iguales a 1 produce $1/\varphi$, donde

φ es la razón áurea (descrita en Section 1.2.2). Una forma de aproximar una fracción continua infinita es truncar la expansión después de un número dado de términos. Tal truncamiento, una llamada *fracción continua finita de* k*-términos*, tiene la forma

$$\cfrac{N_1}{D_1 + \cfrac{N_2}{\ddots + \cfrac{N_k}{D_k}}}\,.$$

Suponga que `n` y `d` son procedimientos de un argumento (el índice del término *i*) que devuelven los N_i y D_i de los términos de la fracción continua. Defina un procedimiento `cont/frac` tal que al evaluar `(cont/frac n d k)` se calcule el valor de la fracción continua finita de *k*-términos. Verifique su procedimiento aproximando $1/\varphi$ usando

```
(cont-frac (lambda (i) 1.0) (lambda (i) 1.0) k)
```

para valores sucesivos de `k`. ¿Qué tan grande debe hacer `k` para obtener una aproximación que sea precisa hasta 4 decimales?

2. Si su procedimiento `cont/frac` genera un proceso recursivo, escriba uno que genere un proceso iterativo. Si genera un proceso iterativo, escriba uno que genere un proceso recursivo.

Ejercicio 1.38: En 1737, el matemático suizo Leonhard Euler publicó una memoria *De Fractionibus Continuis*, que incluía una expansión de fracción continua para $e - 2$, donde e es la base de los logaritmos naturales. En esta fracción, los N_i son todos 1, y los D_i son sucesivamente 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, …. Escriba un programa que utilice su procedimiento `cont/frac` del Ejercicio 1.37 para aproximar e, basándose en la expansión de Euler.

Ejercicio 1.39: Una representación de fracción continua de la función tangente fue publicada en 1770 por el matemático alemán J.H. Lambert:

$$\tan x = \cfrac{x}{1 - \cfrac{x^2}{3 - \cfrac{x^2}{5 - \dots}}}\,,$$

donde x está en radianes. Defina un procedimiento `(tan/cf x k)` que calcule una aproximación a la función tangente basándose en la fórmula de Lambert. `k` especifica el número de términos a calcular, como en el Ejercicio 1.37.

Procedimientos como Valores Devueltos

Los ejemplos anteriores demuestran cómo la capacidad de pasar

procedimientos como argumentos aumenta significativamente el poder expresivo de nuestro lenguaje de programación. Podemos lograr un poder expresivo aún mayor creando procedimientos cuyos valores devueltos sean a su vez procedimientos.

Podemos ilustrar esta idea volviendo a mirar el ejemplo del punto fijo descrito al final de Section 1.3.3. Formulamos una nueva versión del procedimiento de la raíz cuadrada como una búsqueda de punto fijo, comenzando con la observación de que $\sqrt{x}$ es un punto fijo de la función $y \mapsto x/y$. Luego usamos la amortiguación promedio para hacer que las aproximaciones converjan. La amortiguación promedio es una técnica general útil en sí misma. A saber, dada una función *f*, consideramos la función cuyo valor en *x* es igual al promedio de *x* y $f(x)$.

Podemos expresar la idea de amortiguación promedio por medio del siguiente procedimiento:

```
(define (average-damp f) (lambda (x) (average x (f x))))
```

`average/damp` es un procedimiento que toma como argumento un procedimiento `f` y devuelve como valor un procedimiento (producido por el `lambda`) que, cuando se aplica a un número `x`, produce el promedio de `x` y `(f x)`. Por ejemplo, aplicar `average/damp` al procedimiento `square` produce un procedimiento cuyo valor en algún número *x* es el promedio de *x* y x^2. Aplicar este procedimiento resultante a 10 devuelve el promedio de 10 y 100, o 55:[59]

```
((average-damp square) 10)  55
```

Usando `average/damp`, podemos reformular el procedimiento de la raíz cuadrada de la siguiente manera:

```
(define (sqrt x) (fixed-point (average-damp (lambda (y) (/ x y))) 1.0))
```

Observe cómo esta formulación hace explícitas las tres ideas del método: búsqueda de punto fijo, amortiguación promedio y la función $y \mapsto x/y$. Es instructivo comparar esta formulación del método de la raíz cuadrada con la versión original dada en Section 1.1.7. Tenga en cuenta que estos procedimientos expresan el mismo proceso, y observe cuánto más clara se vuelve la idea cuando expresamos el proceso en términos de estas abstracciones. En general, hay muchas formas de formular un proceso como un procedimiento. Los programadores experimentados saben cómo elegir formulaciones procedimentales que sean particularmente perspicuas, y donde los elementos útiles del proceso se exponen como entidades separadas que pueden reutilizarse en otras aplicaciones. Como un simple ejemplo de

reutilización, observe que la raíz cúbica de x es un punto fijo de la función $y \mapsto x/y^2$, por lo que podemos generalizar inmediatamente nuestro procedimiento de raíz cuadrada a uno que extraiga raíces cúbicas:[60]

```
(define (cube-root x) (fixed-point (average-damp (lambda (y) (/ x (square y)))) 1.0))
```

Método de Newton

Cuando introdujimos por primera vez el procedimiento de la raíz cuadrada, en Section 1.1.7, mencionamos que este era un caso especial del *método de Newton*. Si $x \mapsto g(x)$ es una función diferenciable, entonces una solución de la ecuación $g(x) = 0$ es un punto fijo de la función $x \mapsto f(x)$, donde

$$f(x) = x - \frac{g(x)}{Dg(x)}$$

y $Dg(x)$ es la derivada de g evaluada en x. El método de Newton es el uso del método del punto fijo que vimos anteriormente para aproximar una solución de la ecuación encontrando un punto fijo de la función f.[61]

Para muchas funciones g y para suposiciones iniciales suficientemente buenas para x, el método de Newton converge muy rápidamente a una solución de $g(x) = 0$.[62]

Para implementar el método de Newton como un procedimiento, primero debemos expresar la idea de la derivada. Tenga en cuenta que la "derivada", al igual que la amortiguación promedio, es algo que transforma una función en otra función. Por ejemplo, la derivada de la función $x \mapsto x^3$ es la función $x \mapsto 3x^2$. En general, si g es una función y dx es un número pequeño, entonces la derivada Dg de g es la función cuyo valor en cualquier número x viene dado (en el límite de dx pequeño) por

$$Dg(x) = \frac{g(x + dx) - g(x)}{dx}\,.$$

Por lo tanto, podemos expresar la idea de la derivada (tomando dx como, digamos, 0.00001) como el procedimiento

```
(define (deriv g) (lambda (x) (/ (- (g (+ x dx)) (g x)) dx)))
```

junto con la definición

```
(define dx 0.00001)
```

Al igual que `average/damp`, `deriv` es un procedimiento que toma un procedimiento como argumento y devuelve un procedimiento como valor.

Por ejemplo, para aproximar la derivada de $x \mapsto x^3$ en 5 (cuyo valor exacto es 75) podemos evaluar

(define (cube x) (* x x x)) ((deriv cube) 5) *75.00014999664018*

Con la ayuda de `deriv`, podemos expresar el método de Newton como un proceso de punto fijo:

(define (newton-transform g) (lambda (x) (- x (/ (g x) ((deriv g) x))))) (define (newtons-method g guess) (fixed-point (newton-transform g) guess))

El procedimiento `newton/transform` expresa la fórmula al principio de esta sección, y `newtons/method` se define fácilmente en términos de esto. Toma como argumentos un procedimiento que calcula la función para la cual queremos encontrar un cero, junto con una suposición inicial. Por ejemplo, para encontrar la raíz cuadrada de x, podemos usar el método de Newton para encontrar un cero de la función $y \mapsto y^2 - x$ comenzando con una suposición inicial de 1.[63]

Esto proporciona otra forma del procedimiento de la raíz cuadrada:

(define (sqrt x) (newtons-method (lambda (y) (- (square y) x)) 1.0))

Abstracciones y procedimientos de primera clase

Hemos visto dos formas de expresar el cálculo de la raíz cuadrada como una instancia de un método más general, una vez como una búsqueda de punto fijo y otra vez usando el método de Newton. Dado que el método de Newton se expresó a su vez como un proceso de punto fijo, en realidad vimos dos formas de calcular las raíces cuadradas como puntos fijos. Cada método comienza con una función y encuentra un punto fijo de alguna transformación de la función. Podemos expresar esta idea general en sí misma como un procedimiento:

(define (fixed-point-of-transform g transform guess) (fixed-point (transform g) guess))

Este procedimiento muy general toma como argumentos un procedimiento `g` que calcula alguna función, un procedimiento que transforma `g` y una suposición inicial. El resultado devuelto es un punto fijo de la función transformada.

Usando esta abstracción, podemos reformular el primer cálculo de la raíz cuadrada de esta sección (donde buscamos un punto fijo de la versión con amortiguación promedio de $y \mapsto x/y$) como una instancia de este método

general:

(define (sqrt x) (fixed-point-of-transform (lambda (y) (/ x y)) average-damp 1.0))

De manera similar, podemos expresar el segundo cálculo de la raíz cuadrada de esta sección (una instancia del método de Newton que encuentra un punto fijo de la transformación de Newton de $y \mapsto y^2 - x$) como

(define (sqrt x) (fixed-point-of-transform (lambda (y) (- (square y) x)) newton-transform 1.0))

Comenzamos Section 1.3 con la observación de que los procedimientos compuestos son un mecanismo de abstracción crucial, porque nos permiten expresar métodos generales de computación como elementos explícitos en nuestro lenguaje de programación. Ahora hemos visto cómo los procedimientos de orden superior nos permiten manipular estos métodos generales para crear más abstracciones.

Como programadores, debemos estar alerta a las oportunidades de identificar las abstracciones subyacentes en nuestros programas y de construir sobre ellas y generalizarlas para crear abstracciones más potentes. Esto no quiere decir que uno siempre deba escribir programas de la manera más abstracta posible; los programadores expertos saben cómo elegir el nivel de abstracción apropiado para su tarea. Pero es importante poder pensar en términos de estas abstracciones, para que podamos estar listos para aplicarlas en nuevos contextos. La importancia de los procedimientos de orden superior es que nos permiten representar estas abstracciones explícitamente como elementos en nuestro lenguaje de programación, de modo que puedan manejarse como cualquier otro elemento computacional.

En general, los lenguajes de programación imponen restricciones sobre las formas en que se pueden manipular los elementos computacionales. Se dice que los elementos con las menores restricciones tienen un estatus de *primera clase*. Algunos de los "derechos y privilegios" de los elementos de primera clase son:[64]

- Pueden ser nombrados por variables.
- Pueden pasarse como argumentos a procedimientos.
- Pueden devolverse como resultados de procedimientos.
- Pueden incluirse en estructuras de datos.[65]

Lisp, a diferencia de otros lenguajes de programación comunes, otorga a los procedimientos un estatus completo de primera clase. Esto plantea desafíos para una implementación eficiente, pero la ganancia resultante en poder expresivo es enorme.[66]

Ejercicio 1.40: Defina un procedimiento `cubic` que se pueda usar junto con el procedimiento `newtons/method` en expresiones de la forma

(newtons-method (cubic a b c) 1)

para aproximar los ceros del cúbico $x^3 + ax^2 + bx + c$.

Ejercicio 1.41: Defina un procedimiento `double` que tome un procedimiento de un argumento como argumento y devuelva un procedimiento que aplique el procedimiento original dos veces. Por ejemplo, si `inc` es un procedimiento que suma 1 a su argumento, entonces `(double inc)` debería ser un procedimiento que suma 2. ¿Qué valor se devuelve por

(((double (double double)) inc) 5)

Ejercicio 1.42: Sean f y g dos funciones de un argumento. La *composición* f después de g se define como la función $x \mapsto f(g(x))$. Defina un procedimiento `compose` que implemente la composición. Por ejemplo, si `inc` es un procedimiento que suma 1 a su argumento,

((compose square inc) 6) *49*

Ejercicio 1.43: Si f es una función numérica y n es un entero positivo, entonces podemos formar la n^{th} aplicación repetida de f, que se define como la función cuyo valor en x es $f(f(\ldots(f(x))\ldots))$. Por ejemplo, si f es la función $x \mapsto x + 1$, entonces la n^{th} aplicación repetida de f es la función $x \mapsto x + n$. Si f es la operación de elevar un número al cuadrado, entonces la n^{th} aplicación repetida de f es la función que eleva su argumento a la potencia 2^n. Escriba un procedimiento que tome como entradas un procedimiento que calcule f y un entero positivo n y devuelva el procedimiento que calcule la n^{th} aplicación repetida de f. Su procedimiento debería poder usarse de la siguiente manera:

((repeated square 2) 5) *625*

Pista: Puede que le resulte conveniente usar `compose` del Ejercicio 1.42.

Ejercicio 1.44: La idea de *suavizar* una función es un concepto importante en el procesamiento de señales. Si f es una función y dx es un

número pequeño, entonces la versión suavizada de f es la función cuyo valor en un punto x es el promedio de $f(x - dx)$, $f(x)$, y $f(x + dx)$. Escriba un procedimiento `smooth` que tome como entrada un procedimiento que calcule f y devuelva un procedimiento que calcule la f suavizada. A veces es valioso suavizar repetidamente una función (es decir, suavizar la función suavizada, y así sucesivamente) para obtener la *función suavizada* n *veces*. Muestre cómo generar la función suavizada n veces de cualquier función dada usando `smooth` y `repeated` del Ejercicio 1.43.

Ejercicio 1.45: Vimos en Section 1.3.3 que intentar calcular raíces cuadradas encontrando ingenuamente un punto fijo de $y \mapsto x/y$ no converge, y que esto se puede solucionar con amortiguación promedio. El mismo método funciona para encontrar raíces cúbicas como puntos fijos de la $y \mapsto x/y^2$ con amortiguación promedio. Desafortunadamente, el proceso no funciona para las raíces cuartas: una sola amortiguación promedio no es suficiente para hacer que una búsqueda de punto fijo para $y \mapsto x/y^3$ converja. Por otro lado, si aplicamos la amortiguación promedio dos veces (es decir, usamos la amortiguación promedio de la amortiguación promedio de $y \mapsto x/y^3$) la búsqueda de punto fijo converge. Haga algunos experimentos para determinar cuántas amortiguaciones promedio se requieren para calcular las raíces n^{th} como una búsqueda de punto fijo basada en la amortiguación promedio repetida de $y \mapsto x/y^{n-1}$. Use esto para implementar un procedimiento simple para calcular las raíces n^{th} usando `fixed/point`, `average/damp` y el procedimiento `repeated` del Ejercicio 1.43. Suponga que cualquier operación aritmética que necesite está disponible como primitiva.

Ejercicio 1.46: Varios de los métodos numéricos descritos en este capítulo son instancias de una estrategia computacional extremadamente general conocida como *mejora iterativa*. La mejora iterativa dice que, para calcular algo, comenzamos con una suposición inicial para la respuesta, probamos si la suposición es lo suficientemente buena y, de lo contrario, mejoramos la suposición y continuamos el proceso usando la suposición mejorada como la nueva suposición. Escriba un procedimiento `iterative/improve` que tome dos procedimientos como argumentos: un método para decir si una suposición es lo suficientemente buena y un método para mejorar una suposición. `iterative/improve` debería devolver como valor un procedimiento que tome una suposición como argumento y siga mejorando la suposición hasta que sea lo suficientemente buena. Reescriba el procedimiento `sqrt` de Section 1.1.7 y el procedimiento `fixed/point` de Section 1.3.3 en términos de `iterative/improve`.

1. El *Manual del programador de Lisp 1* apareció en 1960, y el *Manual del programador de Lisp 1.5* (McCarthy et al. 1965) se publicó en 1962. La historia temprana de Lisp se describe en McCarthy 1978.

2. Los dos dialectos en los que se escribieron la mayoría de los principales programas de Lisp de la década de 1970 son MacLisp (Moon 1978; Pitman 1983), desarrollado en el Proyecto MAC del MIT, e Interlisp (Teitelman 1974), desarrollado en Bolt Beranek and Newman Inc. y en el Centro de Investigación de Xerox Palo Alto. Portable Standard Lisp (Hearn 1969; Griss 1981) fue un dialecto de Lisp diseñado para ser fácilmente portable entre diferentes máquinas. MacLisp generó varios subdialectos, como Franz Lisp, que se desarrolló en la Universidad de California en Berkeley, y Zetalisp (Moon and Weinreb 1981), que se basaba en un procesador de propósito especial diseñado en el Laboratorio de Inteligencia Artificial del MIT para ejecutar Lisp de manera muy eficiente. El dialecto de Lisp utilizado en este libro, llamado Scheme (Steele and Sussman 1975), fue inventado en 1975 por Guy Lewis Steele Jr. y Gerald Jay Sussman del Laboratorio de Inteligencia Artificial del MIT y posteriormente reimplementado para uso educativo en el MIT. Scheme se convirtió en un estándar del IEEE en 1990 (IEEE 1990). El dialecto Common Lisp (Steele 1982, Steele 1990) fue desarrollado por la comunidad de Lisp para combinar características de los dialectos de Lisp anteriores para crear un estándar industrial para Lisp. Common Lisp se convirtió en un estándar ANSI en 1994 (ANSI 1994).

3. Una de esas aplicaciones especiales fue un cálculo innovador de importancia científica: una integración del movimiento del Sistema Solar que extendió los resultados anteriores en casi dos órdenes de magnitud y demostró que la dinámica del Sistema Solar es caótica. Este cálculo fue posible gracias a nuevos algoritmos de integración, un compilador de propósito especial y un ordenador de propósito especial, todos implementados con la ayuda de herramientas de software escritas en Lisp (Abelson et al. 1992; Sussman and Wisdom 1992).

4. La caracterización de los números como “datos simples” es una fanfarronada descarada. De hecho, el tratamiento de los números es uno de los aspectos más complicados y confusos de cualquier lenguaje de programación. Algunos problemas típicos son estos: algunos sistemas informáticos distinguen entre *enteros*, como 2, y *números reales*, como 2.71. ¿Es el número real 2.00 diferente del entero 2? ¿Son las operaciones aritméticas utilizadas para los enteros las mismas que las operaciones utilizadas para los números reales? ¿6 dividido por 2 produce 3 o 3.0? ¿Qué tan grande puede ser un número que podemos representar? ¿Cuántos decimales de precisión podemos representar? ¿Es el rango de los enteros el mismo que el rango de los números reales? Además de estas preguntas, por

supuesto, yace una colección de problemas relacionados con los errores de redondeo y truncamiento, toda la ciencia del análisis numérico. Dado que nuestro enfoque en este libro está en el diseño de programas a gran escala en lugar de en técnicas numéricas, vamos a ignorar estos problemas. Los ejemplos numéricos de este capítulo mostrarán el comportamiento de redondeo habitual que se observa al usar operaciones aritméticas que conservan un número limitado de decimales de precisión en operaciones no enteras.

5. A lo largo de este libro, cuando deseemos enfatizar la distinción entre la entrada escrita por el usuario y la respuesta impresa por el intérprete, mostraremos esta última en caracteres inclinados.

6. Los sistemas Lisp suelen proporcionar características para ayudar al usuario a formatear expresiones. Dos características especialmente útiles son una que indenta automáticamente a la posición de pretty-print adecuada cada vez que se inicia una nueva línea y otra que resalta el paréntesis izquierdo correspondiente cada vez que se escribe un paréntesis derecho.

7. Lisp obedece la convención de que toda expresión tiene un valor. Esta convención, junto con la antigua reputación de Lisp como un lenguaje ineficiente, es la fuente de la ocurrencia de Alan Perlis (parafraseando a Oscar Wilde) de que “los programadores de Lisp conocen el valor de todo pero el costo de nada”.

8. En este libro, no mostramos la respuesta del intérprete a la evaluación de definiciones, ya que esto depende mucho de la implementación.

9. Capítulo 3 mostrará que esta noción de entorno es crucial, tanto para entender cómo funciona el intérprete como para implementar intérpretes.

10. Puede parecer extraño que la regla de evaluación diga, como parte del primer paso, que debemos evaluar el elemento más a la izquierda de una combinación, ya que en este punto eso solo puede ser un operador como `+` o `*` que representa un procedimiento primitivo incorporado como la suma o la multiplicación. Veremos más adelante que es útil poder trabajar con combinaciones cuyos operadores son en sí mismos expresiones compuestas.

11. Las formas sintácticas especiales que son simplemente estructuras superficiales alternativas convenientes para cosas que se pueden escribir de maneras más uniformes a veces se llaman *azúcar sintáctico*, para usar una frase acuñada por Peter Landin. En comparación con los usuarios de otros lenguajes, los programadores de Lisp, por regla general, están menos preocupados por cuestiones de sintaxis. (Por el contrario, examine cualquier manual de Pascal y observe cuánto se dedica a las descripciones de la

sintaxis). Este desdén por la sintaxis se debe en parte a la flexibilidad de Lisp, que facilita el cambio de la sintaxis superficial, y en parte a la observación de que muchas construcciones sintácticas "convenientes", que hacen que el lenguaje sea menos uniforme, terminan causando más problemas de los que valen cuando los programas se vuelven grandes y complejos. En palabras de Alan Perlis, "El azúcar sintáctico causa cáncer de punto y coma".

12. Observe que aquí se están combinando dos operaciones diferentes: estamos creando el procedimiento y le estamos dando el nombre `square`. Es posible, de hecho importante, poder separar estas dos nociones: crear procedimientos sin nombrarlos y dar nombres a procedimientos que ya han sido creados. Veremos cómo hacer esto en Section 1.3.2.

13. A lo largo de este libro, describiremos la sintaxis general de las expresiones usando símbolos en cursiva delimitados por corchetes angulares, p. ej., ⟨*nombre*$\kern0.08em\rangle$, para denotar los "espacios" en la expresión que se deben rellenar cuando se usa realmente dicha expresión.

14. De manera más general, el cuerpo del procedimiento puede ser una secuencia de expresiones. En este caso, el intérprete evalúa cada expresión de la secuencia a su vez y devuelve el valor de la expresión final como el valor de la aplicación del procedimiento.

15. A pesar de la simplicidad de la idea de sustitución, resulta sorprendentemente complicado dar una definición matemática rigurosa del proceso de sustitución. El problema surge de la posibilidad de confusión entre los nombres utilizados para los parámetros formales de un procedimiento y los nombres (posiblemente idénticos) utilizados en las expresiones a las que se puede aplicar el procedimiento. De hecho, hay una larga historia de definiciones erróneas de *sustitución* en la literatura de la lógica y la semántica de la programación. Consulte Stoy 1977 para una discusión cuidadosa de la sustitución.

16. En el Capítulo 3 introduciremos el *procesamiento de flujos*, que es una forma de manejar estructuras de datos aparentemente "infinitas" incorporando una forma limitada de evaluación en orden normal. En Section 4.2 modificaremos el intérprete de Scheme para producir una variante de Scheme de orden normal.

17. “Interpretado como verdadero o falso” significa lo siguiente: en Scheme, hay dos valores distinguidos que se denotan con las constantes `#t` y `#f`. Cuando el intérprete comprueba el valor de un predicado, interpreta `#f` como falso. Cualquier otro valor se trata como verdadero. (Por lo tanto, proporcionar `#t` es lógicamente innecesario, pero es conveniente). En este

libro usaremos los nombres `true` y `false`, que están asociados con los valores `#t` y `#f` respectivamente.

18. `abs` también utiliza el operador "menos" `-`, que, cuando se usa con un solo operando, como en `(- x)`, indica negación.

19. Una diferencia menor entre `if` y `cond` es que la parte ⟨*e*⟩ de cada cláusula `cond` puede ser una secuencia de expresiones. Si se encuentra que el ⟨*p*⟩ correspondiente es verdadero, las expresiones ⟨*e*⟩ se evalúan en secuencia y el valor de la expresión final en la secuencia se devuelve como el valor del `cond`. En una expresión `if`, sin embargo, el ⟨*consecuente*$\kern0.04em\rangle $ y la ⟨*alternativa*$\kern0.04em\rangle$ deben ser expresiones únicas.

20. Las descripciones declarativas e imperativas están íntimamente relacionadas, como de hecho lo están las matemáticas y la informática. Por ejemplo, decir que la respuesta producida por un programa es "correcta" es hacer una declaración declarativa sobre el programa. Hay una gran cantidad de investigación destinada a establecer técnicas para probar que los programas son correctos, y gran parte de la dificultad técnica de este tema tiene que ver con negociar la transición entre las declaraciones imperativas (a partir de las cuales se construyen los programas) y las declaraciones declarativas (que se pueden usar para deducir cosas). En una línea relacionada, un área actual importante en el diseño de lenguajes de programación es la exploración de los llamados lenguajes de muy alto nivel, en los que uno realmente programa en términos de declaraciones declarativas. La idea es hacer que los intérpretes sean lo suficientemente sofisticados para que, dado el conocimiento de "lo que es" especificado por el programador, puedan generar conocimiento de "cómo hacer" automáticamente. Esto no se puede hacer en general, pero hay áreas importantes donde se ha progresado. Volveremos a esta idea en el Capítulo 4.

21. Este algoritmo de raíz cuadrada es en realidad un caso especial del método de Newton, que es una técnica general para encontrar raíces de ecuaciones. El algoritmo de raíz cuadrada en sí fue desarrollado por Herón de Alejandría en el siglo I d.C. Veremos cómo expresar el método general de Newton como un procedimiento Lisp en Section 1.3.4.

22. Generalmente daremos a los predicados nombres que terminen con signos de interrogación, para ayudarnos a recordar que son predicados. Esto es solo una convención estilística. En lo que respecta al intérprete, el signo de interrogación es solo un carácter ordinario.

23. Observe que expresamos nuestra suposición inicial como 1.0 en lugar de 1. Esto no haría ninguna diferencia en muchas implementaciones de Lisp.

MIT Scheme, sin embargo, distingue entre enteros exactos y valores decimales, y dividir dos enteros produce un número racional en lugar de un decimal. Por ejemplo, dividir 10 por 6 produce 5/3, mientras que dividir 10.0 por 6.0 produce 1.6666666666666667. (Aprenderemos a implementar la aritmética en números racionales en Section 2.1.1.) Si comenzamos con una suposición inicial de 1 en nuestro programa de raíz cuadrada, y *x* es un entero exacto, todos los valores posteriores producidos en el cálculo de la raíz cuadrada serán números racionales en lugar de decimales. Las operaciones mixtas en números racionales y decimales siempre producen decimales, por lo que comenzar con una suposición inicial de 1.0 fuerza a que todos los valores posteriores sean decimales.

24. Los lectores que estén preocupados por los problemas de eficiencia involucrados en el uso de llamadas a procedimientos para implementar la iteración deben tener en cuenta las observaciones sobre la "recursión de cola" en Section 1.2.1.

25. Ni siquiera está claro cuál de estos procedimientos es una implementación más eficiente. Esto depende del hardware disponible. Hay máquinas para las cuales la implementación "obvia" es la menos eficiente. Considere una máquina que tiene extensas tablas de logaritmos y antilogaritmos almacenadas de una manera muy eficiente.

26. El concepto de renombramiento consistente es en realidad sutil y difícil de definir formalmente. Lógicos famosos han cometido errores embarazosos aquí.

27. El alcance léxico dicta que las variables libres en un procedimiento se toman para referirse a los enlaces hechos por las definiciones de procedimientos envolventes; es decir, se buscan en el entorno en el que se definió el procedimiento. Veremos cómo funciona esto en detalle en el capítulo 3 cuando estudiemos los entornos y el comportamiento detallado del intérprete.

28. Las definiciones incrustadas deben ir primero en el cuerpo de un procedimiento. La administración no se hace responsable de las consecuencias de ejecutar programas que entrelazan definición y uso.

29. En un programa real probablemente usaríamos la estructura de bloques introducida en la última sección para ocultar la definición de `fact/iter`:

(define (factorial n) (define (iter product counter) (if (> counter n) product (iter (* counter product) (+ counter 1)))) (iter 1 1))

Evitamos hacerlo aquí para minimizar el número de cosas en las que pensar a

la vez.

30. Cuando discutamos la implementación de procedimientos en máquinas de registros en el Capítulo 5, veremos que cualquier proceso iterativo puede realizarse "en hardware" como una máquina que tiene un conjunto fijo de registros y no necesita memoria auxiliar. En contraste, realizar un proceso recursivo requiere una máquina que use una estructura de datos auxiliar conocida como *pila*.

31. La recursión de cola se conoce desde hace mucho tiempo como un truco de optimización del compilador. Carl Hewitt (1977) proporcionó una base semántica coherente para la recursión de cola, quien la explicó en términos del modelo de "paso de mensajes" de la computación que discutiremos en el Capítulo 3. Inspirados por esto, Gerald Jay Sussman y Guy Lewis Steele Jr. (ver Steele and Sussman 1975) construyeron un intérprete recursivo de cola para Scheme. Steele mostró más tarde cómo la recursión de cola es una consecuencia de la forma natural de compilar las llamadas a procedimientos (Steele 1977). El estándar del IEEE para Scheme requiere que las implementaciones de Scheme sean recursivas de cola.

32. Un ejemplo de esto se insinuó en Section 1.1.3. El propio intérprete evalúa expresiones usando un proceso recursivo de árbol.

33. Por ejemplo, trabaje en detalle cómo se aplica la regla de reducción al problema de dar cambio de 10 centavos usando centavos y monedas de cinco centavos.

34. Un enfoque para hacer frente a los cálculos redundantes es organizar las cosas de modo que construyamos automáticamente una tabla de valores a medida que se calculan. Cada vez que se nos pide que apliquemos el procedimiento a algún argumento, primero miramos si el valor ya está almacenado en la tabla, en cuyo caso evitamos realizar el cálculo redundante. Esta estrategia, conocida como *tabulación* o *memoización*, se puede implementar de manera sencilla. La tabulación a veces se puede utilizar para transformar procesos que requieren un número exponencial de pasos (como `count/change`) en procesos cuyos requisitos de espacio y tiempo crecen linealmente con la entrada. Ver Ejercicio 3.27.

35. Los elementos del triángulo de Pascal se llaman *coeficientes binomiales*, porque la n^{th} fila consiste en los coeficientes de los términos en la expansión de $(x + y)^n$. Este patrón para calcular los coeficientes apareció en la obra seminal de Blaise Pascal de 1653 sobre la teoría de la probabilidad, *Traité du triangle arithmétique*. Según Knuth (1973), el mismo patrón aparece en el *Szu-yuen Yü-chien* (“El Precioso Espejo de los Cuatro Elementos”), publicado por

el matemático chino Chu Shih-chieh en 1303, en las obras del poeta y matemático persa del siglo XII Omar Khayyam, y en las obras del matemático hindú del siglo XII Bháscara Áchárya.

36. Estas afirmaciones ocultan una gran simplificación. Por ejemplo, si contamos los pasos del proceso como "operaciones de máquina", estamos asumiendo que el número de operaciones de máquina necesarias para realizar, digamos, una multiplicación es independiente del tamaño de los números a multiplicar, lo cual es falso si los números son suficientemente grandes. Observaciones similares se aplican a las estimaciones de espacio. Al igual que el diseño y la descripción de un proceso, el análisis de un proceso se puede llevar a cabo en varios niveles de abstracción.

37. Más precisamente, el número de multiplicaciones requeridas es igual a 1 menos que el logaritmo en base 2 de n más el número de unos en la representación binaria de n. Este total siempre es menor que el doble del logaritmo en base 2 de n. Las constantes arbitrarias k_1 y k_2 en la definición de la notación de orden implican que, para un proceso logarítmico, la base a la que se toman los logaritmos no importa, por lo que todos esos procesos se describen como $\Theta(\log n)$.

38. Quizás se pregunte por qué a alguien le importaría elevar números a la potencia 1000. Consulte Section 1.2.6.

39. Este algoritmo iterativo es antiguo. Aparece en el *Chandah-sutra* de Áchárya Pingala, escrito antes del 200 a.C. Consulte Knuth 1981, sección 4.6.3, para una discusión y análisis completos de este y otros métodos de exponenciación.

40. Este algoritmo, que a veces se conoce como el "método del campesino ruso" de multiplicación, es antiguo. Se encuentran ejemplos de su uso en el Papiro de Rhind, uno de los dos documentos matemáticos más antiguos que existen, escrito alrededor del 1700 a.C. (y copiado de un documento aún más antiguo) por un escriba egipcio llamado A'h-mose.

41. Este ejercicio nos fue sugerido por Joe Stoy, basado en un ejemplo en Kaldewaij 1990.

42. El Algoritmo de Euclides se llama así porque aparece en los *Elementos* de Euclides (Libro 7, ca. 300 a.C.). Según Knuth (1973), puede considerarse el algoritmo no trivial más antiguo conocido. El método egipcio antiguo de multiplicación (Ejercicio 1.18) es seguramente más antiguo, pero, como explica Knuth, el algoritmo de Euclides es el más antiguo conocido que se ha presentado como un algoritmo general, en lugar de como un conjunto de ejemplos ilustrativos.

43. Este teorema fue demostrado en 1845 por Gabriel Lamé, un matemático e ingeniero francés conocido principalmente por sus contribuciones a la física matemática. Para demostrar el teorema, consideramos pares (a_k, b_k), donde $a_k \geq b_k$, para los cuales el Algoritmo de Euclides termina en k pasos. La prueba se basa en la afirmación de que, si $(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$ son tres pares sucesivos en el proceso de reducción, entonces debemos tener $b_{k+1} \geq b_k + b_{k-1}$. Para verificar la afirmación, considere que un paso de reducción se define aplicando la transformación $a_{k-1} = b_k$, $b_{k-1} =$ resto de a_k dividido por b_k. La segunda ecuación significa que $a_k = qb_k + b_{k-1}$ para algún entero positivo q. Y como q debe ser al menos 1, tenemos $a_k = qb_k + b_{k-1} \geq b_k + b_{k-1}$. Pero en el paso de reducción anterior tenemos $b_{k+1} = a_k$. Por lo tanto, $b_{k+1} = a_k \geq b_k + b_{k-1}$. Esto verifica la afirmación. Ahora podemos demostrar el teorema por inducción sobre k, el número de pasos que el algoritmo requiere para terminar. El resultado es verdadero para $k = 1$, ya que esto simplemente requiere que b sea al menos tan grande como Fib(1) = 1. Ahora, suponga que el resultado es verdadero para todos los enteros menores o iguales a k y establezca el resultado para $k + 1$. Sean $(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$ pares sucesivos en el proceso de reducción. Por nuestras hipótesis de inducción, tenemos $b_{k-1} \ge {\rm Fib}(k - 1)$ y $b_k \ge {\rm Fib}(k)$. Así, aplicando la afirmación que acabamos de probar junto con la definición de los números de Fibonacci, obtenemos $b_{k+1} \ge b_k + b_{k-1} \ge {\rm Fib}(k) + {\rm Fib}(k-1) = {\rm Fib}(k+1)$, lo que completa la prueba del Teorema de Lamé.

44. Si d es un divisor de n, entonces también lo es n/d. Pero d y n/d no pueden ser ambos mayores que $\sqrt{n}$.

45. Pierre de Fermat (1601-1665) es considerado el fundador de la teoría de números moderna. Obtuvo muchos resultados importantes en teoría de números, pero usualmente solo anunciaba los resultados, sin proporcionar sus pruebas. El Pequeño Teorema de Fermat fue enunciado en una carta que escribió en 1640. La primera prueba publicada fue dada por Euler en 1736 (y una prueba anterior, idéntica, fue descubierta en los manuscritos no publicados de Leibniz). El más famoso de los resultados de Fermat — conocido como el Último Teorema de Fermat— fue anotado en 1637 en su copia del libro *Aritmética* (del matemático griego del siglo III Diofanto) con la observación «He descubierto una prueba verdaderamente notable, pero este margen es demasiado pequeño para contenerla». Encontrar una prueba del Último Teorema de Fermat se convirtió en uno de los desafíos más famosos en la teoría de números. Una solución completa fue finalmente dada en 1995 por Andrew Wiles de la Universidad de Princeton.

46. Los pasos de reducción en los casos en que el exponente e es mayor que 1

se basan en el hecho de que, para cualquier entero *x*, *y* y *m*, podemos encontrar el resto de *x* por *y* módulo *m* calculando por separado los restos de *x* módulo *m* y *y* módulo *m*, multiplicándolos y luego tomando el resto del resultado módulo *m*. Por ejemplo, en el caso en que *e* es par, calculamos el resto de $b^{e/2}$ módulo *m*, lo elevamos al cuadrado y tomamos el resto módulo *m*. Esta técnica es útil porque significa que podemos realizar nuestro cálculo sin tener que lidiar con números mucho más grandes que *m*. (Compare con el Ejercicio 1.25).

47. Los números que engañan al test de Fermat se llaman *números de Carmichael*, y se sabe poco sobre ellos, aparte de que son extremadamente raros. Hay 255 números de Carmichael por debajo de 100,000,000. Los más pequeños son 561, 1105, 1729, 2465, 2821 y 6601. Al probar la primalidad de números muy grandes elegidos al azar, la posibilidad de tropezar con un valor que engañe al test de Fermat es menor que la posibilidad de que la radiación cósmica haga que el ordenador cometa un error al ejecutar un algoritmo "correcto". Considerar que un algoritmo es inadecuado por la primera razón pero no por la segunda ilustra la diferencia entre las matemáticas y la ingeniería.

48. Una de las aplicaciones más sorprendentes de las pruebas de primalidad probabilísticas ha sido en el campo de la criptografía. Aunque ahora es computacionalmente inviable factorizar un número arbitrario de 200 dígitos, la primalidad de dicho número se puede verificar en unos pocos segundos con el test de Fermat. Este hecho forma la base de una técnica para construir "códigos inquebrantables" sugerida por Rivest et al. (1977). El *algoritmo RSA* resultante se ha convertido en una técnica ampliamente utilizada para mejorar la seguridad de las comunicaciones electrónicas. Debido a esto y a desarrollos relacionados, el estudio de los números primos, una vez considerado el epítome de un tema en matemáticas "puras" que se estudiaba solo por sí mismo, ahora resulta tener importantes aplicaciones prácticas en la criptografía, la transferencia electrónica de fondos y la recuperación de información.

49. Esta serie, generalmente escrita en la forma equivalente ${\pi\over4} = 1 - {1\over3} + {1\over5} - {1\over7} + \dots$, se debe a Leibniz. Veremos cómo usar esto como base para algunos trucos numéricos sofisticados en Section 3.5.3.

50. Observe que hemos usado la estructura de bloques (Section 1.1.8) para incrustar las definiciones de `pi/next` y `pi/term` dentro de `pi/sum`, ya que es poco probable que estos procedimientos sean útiles para cualquier otro propósito. Veremos cómo deshacernos de ellos por completo en Section 1.3.2.

51. La intención de los Ejercicio 1.31 a Ejercicio 1.33 es demostrar el poder expresivo que se logra al usar una abstracción apropiada para consolidar muchas operaciones aparentemente dispares. Sin embargo, aunque la acumulación y el filtrado son ideas elegantes, nuestras manos están algo atadas al usarlas en este punto, ya que todavía no tenemos estructuras de datos para proporcionar medios de combinación adecuados para estas abstracciones. Volveremos a estas ideas en Section 2.2.3 cuando mostremos cómo usar *secuencias* como interfaces para combinar filtros y acumuladores para construir abstracciones aún más potentes. Veremos allí cómo estos métodos realmente se destacan como un enfoque potente y elegante para diseñar programas.

52. Esta fórmula fue descubierta por el matemático inglés del siglo XVII John Wallis.

53. Sería más claro y menos intimidante para las personas que aprenden Lisp si se usara un nombre más obvio que `lambda`, como `make/procedure`. Pero la convención está firmemente arraigada. La notación se adopta del cálculo λ, un formalismo matemático introducido por el lógico matemático Alonzo Church (1941). Church desarrolló el cálculo λ para proporcionar una base rigurosa para estudiar las nociones de función y aplicación de funciones. El cálculo λ se ha convertido en una herramienta básica para las investigaciones matemáticas de la semántica de los lenguajes de programación.

54. Comprender las definiciones internas lo suficientemente bien como para estar seguro de que un programa significa lo que pretendemos que signifique requiere un modelo más elaborado del proceso de evaluación de lo que hemos presentado en este capítulo. Sin embargo, las sutilezas no surgen con las definiciones internas de procedimientos. Volveremos a este tema en Section 4.1.6, después de aprender más sobre la evaluación.

55. Hemos usado 0.001 como un número "pequeño" representativo para indicar una tolerancia para el error aceptable en un cálculo. La tolerancia apropiada para un cálculo real depende del problema a resolver y de las limitaciones del ordenador y del algoritmo. Esto suele ser una consideración muy sutil, que requiere la ayuda de un analista numérico o algún otro tipo de mago.

56. Esto se puede lograr usando `error`, que toma como argumentos una serie de elementos que se imprimen como mensajes de error.

57. Pruebe esto durante una clase aburrida: ponga su calculadora en modo de radianes y luego presione repetidamente el botón `cos` hasta que obtenga el punto fijo.

58. ↦ (pronunciado "mapea a") es la forma del matemático de escribir `lambda`. $y \mapsto x/y$ significa `(lambda (y) (/ x y))`, es decir, la función cuyo valor en y es x/y.

59. Observe que esta es una combinación cuyo operador es a su vez una combinación. El Ejercicio 1.4 ya demostró la capacidad de formar tales combinaciones, pero ese fue solo un ejemplo de juguete. Aquí comenzamos a ver la necesidad real de tales combinaciones: al aplicar un procedimiento que se obtiene como el valor devuelto por un procedimiento de orden superior.

60. Consulte el Ejercicio 1.45 para una generalización adicional.

61. Los libros de cálculo elemental suelen describir el método de Newton en términos de la secuencia de aproximaciones $x_{n+1} = x_n - g(x_n)/Dg(x_n)$. Tener un lenguaje para hablar de procesos y usar la idea de puntos fijos simplifica la descripción del método.

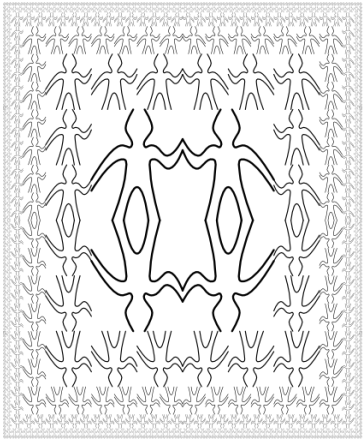
62. El método de Newton no siempre converge a una respuesta, pero se puede demostrar que en casos favorables cada iteración duplica el número de dígitos de precisión de la aproximación a la solución. En tales casos, el método de Newton convergerá mucho más rápidamente que el método de la bisección.

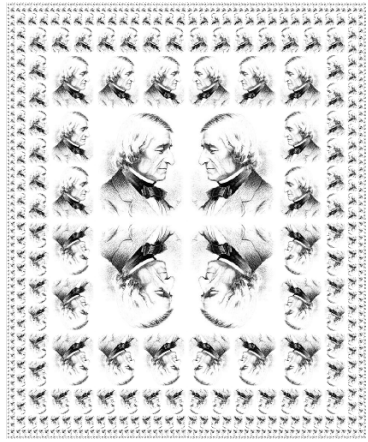
63. Para encontrar raíces cuadradas, el método de Newton converge rápidamente a la solución correcta desde cualquier punto de partida.

64. La noción de estatus de primera clase de los elementos del lenguaje de programación se debe al informático británico Christopher Strachey (1916-1975).

65. Veremos ejemplos de esto después de introducir las estructuras de datos en el Capítulo 2.

66. El principal costo de implementación de los procedimientos de primera clase es que permitir que los procedimientos se devuelvan como valores requiere reservar almacenamiento para las variables libres de un procedimiento incluso mientras el procedimiento no se está ejecutando. En la implementación de Scheme que estudiaremos en Section 4.1, estas variables se almacenan en el entorno del procedimiento.





Construcción de Abstracciones con datos

> Llegamos ahora al paso decisivo de la abstracción matemática: nos olvidamos de lo que representan los símbolos. …[El matemático] no tiene por qué estar ocioso; hay muchas operaciones que puede realizar con estos símbolos, sin tener que mirar nunca las cosas que representan.
>
> —Hermann Weyl, *El modo de pensar matemático*

En el Capítulo 1 nos concentramos en los procesos computacionales y en el rol que cumplen los procedimientos en la tarea de diseño de programas. Hemos visto cómo usar datos primitivos (números) y operaciones primitivas (operaciones aritméticas), cómo combinar procedimientos para formar procedimientos compuestos, mediante composición, condicionales, y el uso de parámetros, y cómo abstraer procedimientos mediante `define`. Vimos que se puede considerar a un procedimiento como el patrón de la evolución local de un proceso, y entonces estudiamos algunos patrones comunes de procesos creados por procedimientos: clasificamos los patrones, razonamos acerca de ellos y realizamos simples análisis algorítmicos. Vimos también que los procedimientos de orden superior aumentan la potencia de nuestro lenguaje pues nos permiten manipular, y por lo tanto razonar en términos de, métodos generales de computación. Esta es gran parte de la esencia de la programación.

En este capítulo vamos a usar datos más complejos. Todos los procedimientos en el capítulo 1 operan sobre datos numéricos simples, y los datos simples no son suficientes para muchos de los problemasque quisiéramos tratar mediante una computación. Los programas se diseñan típicamente para modelar fenómenos complejos, y con mucha frecuencia uno construye objetos computacionales compuestos por muchas partes, para modelar fenómenos del mundo real que tienen muchos aspectos. Por lo tanto, si bien nuestro enfoque en el capítulo 1 estaba en la construcción de abstracciones mediante la combinación de procedimientos para formar procedimientos compuestos, en el capítulo actual el objetivo será otro aspecto clave de todo lenguaje de programación: los medios que brinda para construir

abstracciones mediante la combinación de objetos de datos para formar *datos compuestos*.

¿Porqué queremos datos compuestos en un lenguaje de programación? Por las mismas razones que queremos procedimientos compuestos: para elevar el nivel conceptual en el cual podemos diseñar nuestros programas, para aumentar la modularidad de nuestros diseños, y para mejorar el poder expresivo de nuestro lenguaje. Así como la capacidad de definir procedimientos nos permite trabajar con procesos en un nivel conceptual mayor que el provisto por las operaciones primitivas del lenguaje, la capacidad de construir objetos de datos compuestos nos permite trabajar con datos en un nivel conceptual mayor que el de los objetos primitivos de datos del lenguaje.

Considere la tarea de diseñar un sistema para realizar aritmética con números racionales. Podemos imaginar una operación `add/rat` que toma dos números racionales y produce su suma. En términos de datos simples, podemos pensar un número racional como dos enteros: un numerador y un denominador. De esa manera, podemos diseñar un programa en el cual cada número racional sería representado mediante dos enteros (un numerador y un denominador) y donde `add/rat` se podría implementar mediante dos procedimientos (uno produce el numerador de la suma y otro que produce el denominador). Pero esto sería incómodo, porque de ese modo deberíamos llevar la cuenta en forma explícita de cual numerador corresponde a tal denominador. En un sistema pensado para realizar muchas operaciones sobre muchos números racionales, el registro de dichos detalles introduciría una confusión sustancial en los programas, sin mencionar lo que produciría en nuestras mentes. Sería mucho mejor si pudiésemos "entrelazar" un numerador con un denominador para formar un par—un *objeto de datos compuesto*—que pueda ser manipulado por nuestros programas de manera consistente con la idea de un número racional como unidad conceptual de una sola pieza.

El uso de datos compuestos también permite que aumentemos la modularidad de nuestros programas. Si podemos manipular números racionales directamente como objetos por derecho propio, entonces podemos separar la parte de nuestro programa que usa números racionales, de los detalles relacionados con la representación de los números racionales mediante un par de enteros. La técnica general de aislar las partes de un programa que trabajan con los detalles de la representación de los objetos, de las partes del programa que usan esos objetos de datos se denomina *abstracción de datos* y es una poderosa metodología de diseño. Veremos cómo la abstracción de datos hace que los programas sean mucho más fáciles de diseñar, mantener y modificar.

El uso de datos compuestos favorece un aumento real en la potencia expresiva de nuestro lenguaje de programación. Considere la idea de formar una "combinación lineal" *ax + by*. Nos gustaría escribir un procedimiento que acepte *a*, *b*, *x*, y *y* como argumentos y que devuelva el valor de *ax + by*. No hay dificultades para hacer esto si los argumentos son números, porque podemos definir directamente el procedimiento:

```
(define (linear-combination a b x y) (+ (* a x) (* b y)))
```

Ahora bien, supongamos que nos interesa trabajar no sólo con números. Supongamos que nos gustaría expresar, en términos procedurales, la idea de que uno puede formar combinaciones lineales siempre que tengamos definidas la adición y la multiplicación—para números racionales, números complejos, polinomios, etc. Podríamos expresarlo como procedimiento de la forma:

```
(define (linear-combination a b x y) (add (mul a x) (mul b y)))
```

donde `add` y `mul` no son los procedimientos primitivos `+` y `*` sino cosas más complejas que van a realizar las operaciones que sean adecuadas para cualquier tipo de datos que pasemos como los argumentos `a`, `b`, `x`, y `y`. El punto central es que lo único que necesita conocer `linear/combination` sobre `a`, `b`, `x`, y `y` es que los procedimientos `add` y `mul` van a realizar las manipulaciones apropiadas. Desde la perspectiva del procedimiento `linear/combination`, resulta irrelevante que son `a`, `b`, `x`, y `y` y más irrelevante aún cómo es que se los representa en términos de datos más primitivos. Este ejemplo muestra porque es importante que nuestro lenguaje de programación proporcione la capacidad de manipular objetos compuestos directamente: sino fuera así, un procedimiento como `linear/combination` no tiene manera de pasar sus argumentos a `add` y `mul` sin conocer los detalles de su estructura.[1]

Al principio de este capítulo implementaremos el sistema aritmético de números racionales que se menciona más arriba. El mismo será el trasfondo para nuestra discusión sobre datos compuestos y abstracción de datos. Así como trabajamos con los procedimientos compuestos, el tema principal que estudiaremos será la abstracción como técnica para lidiar con la complejidad, y vamos a ver cómo la abstracción de datos nos permite erigir *barreras de abstracción* entre las diferentes partes de un programa.

Veremos que la clave para armar datos compuestos reside en que el lenguaje de programación nos debe proporcionar alguna clase de "pegamento" de manera de formar objetos compuestos más complejos, a partir de la combinación de objetos. Existen varias clases posibles de pegamento. Por

cierto, descubriremos cómo formar datos compuestos sin necesidad de utilizar operaciones de “datos”, sino sólo procedimientos. Con esto vamos a difuminar aún más la distinción entre “procedimientos” y “datos”, la cual ya era tenue al final del capítulo 1. También vamos a explorar algunas técnicas convencionales para representar secuencias y árboles. Una idea clave a la hora de trabajar con datos compuestos es la noción de *clausura*—que el pegamento usado para combinar objetos de datos nos debe permitir combinar no sólo objetos de datos primitivos, sino también objetos compuestos. Otra idea clave es que los objetos de datos compuestos pueden servir como *interfaces convencionales* para combinar módulos de programa de diferentes maneras. Para ilustrar algunas de estas ideas se presenta un lenguaje de gráficos que explota el concepto de *clausura*.

Luego aumentaremos la potencia de representación de nuestro lenguaje mediante la introducción de *expresiones simbólicas*—datos cuyas partes elementales pueden ser símbolos arbitrarios, y no sólo números. Exploraremos varias alternativas para representar conjuntos de objetos. Encontraremos que, así como en el caso de una función numérica dada, ésta puede calcularse mediante diferentes procesos computacionales, hay varias maneras de representar una estructura de datos dada, en términos de objetos más simples, y que la elección de la representación puede impactar significativamente en los requerimientos de tiempo y espacio de los procesos que manipulan esos datos. Investigaremos esas ideas en el contexto de la obtención simbólica de las derivadas, la representación de conjuntos y la codificación de la información.

A continuación nos concentraremos en el problema de trabajar con datos que pueden estar representados de manera diferente en diferentes partes del programa. Esto nos lleva a la necesidad de implementar *operaciones genéricas*, que deben poder manejar diferentes tipos de datos. Para mantener la modularidad en presencia de operaciones genéricas se requieren barreras de abstracción más poderosas de las erigidas sólo con simples abstracciones de datos. En particular, vamos a introducir la técnica denominada *programación dirigida por datos* que permite que cada representación individual de datos se diseñe de manera aislada y luego se combine de manera *aditiva* (o sea, sin modificación). Para ilustrar la potencia de este enfoque al diseño de sistemas, cerramos el capítulo con la aplicación de lo aprendido en la implementación de un paquete para realizar aritmética simbólica sobre polinomios, cuyos coeficientes pueden ser enteros, números racionales, números complejos e incluso otros polinomios.

Introducción a la abstracción de datos

En Section 1.1.8, observamos que un procedimiento utilizado como elemento en la creación de un procedimiento más complejo podría considerarse no solo como una colección de operaciones particulares, sino también como una abstracción procedimental. Es decir, los detalles de cómo se implementó el procedimiento podrían suprimirse, y el procedimiento particular en sí podría reemplazarse por cualquier otro procedimiento con el mismo comportamiento general. En otras palabras, podríamos hacer una abstracción que separara la forma en que se usaría el procedimiento de los detalles de cómo se implementaría el procedimiento en términos de procedimientos más primitivos. La noción análoga para datos compuestos se llama *abstracción de datos*. La abstracción de datos es una metodología que nos permite aislar cómo se utiliza un objeto de datos compuesto de los detalles de cómo se construye a partir de objetos de datos más primitivos.

La idea básica de la abstracción de datos es estructurar los programas que van a usar objetos de datos compuestos para que operen sobre "datos abstractos". Es decir, nuestros programas deben usar los datos de tal manera que no hagan suposiciones sobre los datos que no sean estrictamente necesarias para realizar la tarea en cuestión. Al mismo tiempo, se define una representación de datos "concreta" independientemente de los programas que usan los datos. La interfaz entre estas dos partes de nuestro sistema será un conjunto de procedimientos, llamados *selectores* y *constructores*, que implementan los datos abstractos en términos de la representación concreta. Para ilustrar esta técnica, consideraremos cómo diseñar un conjunto de procedimientos para manipular números racionales.

Ejemplo: Operaciones aritméticas con números racionales

Supongamos que queremos hacer aritmética con números racionales. Queremos poder sumarlos, restarlos, multiplicarlos y dividirlos, y probar si dos números racionales son iguales.

Comencemos asumiendo que ya tenemos una forma de construir un número racional a partir de un numerador y un denominador. También asumimos que, dado un número racional, tenemos una forma de extraer (o seleccionar) su numerador y su denominador. Asumamos además que el constructor y los selectores están disponibles como procedimientos:

- $\hbox{\tt(make-rat}\;\langle{n}\rangle\;\langle{d}\kern0.06em \rangle\hbox{\tt)}$ devuelve el número racional cuyo numerador es el entero $\langle n \rangle$ y cuyo denominador es el entero $\langle{d}\kern0.06em \rangle$.

- $\hbox{\tt(numer}\;\;\langle{x}\rangle\hbox{\tt)}$ devuelve el numerador del número racional ⟨*x*⟩.

- $\hbox{\tt(denom}\;\;\langle{x}\rangle\hbox{\tt)}$ devuelve el denominador del número racional ⟨*x*⟩.

Estamos usando aquí una poderosa estrategia de síntesis: *pensamiento desiderativo*. Aún no hemos dicho cómo se representa un número racional, ni cómo deben implementarse los procedimientos `numer`, `denom` y `make/rat`. Aun así, si tuviéramos estos tres procedimientos, podríamos sumar, restar, multiplicar, dividir y probar la igualdad usando las siguientes relaciones:

$$\begin{aligned} {n_1 \over d_1} + {n_2 \over d_2} &= {n_1 d_2 + n_2 d_1 \over d_1 d_2}, \\ {n_1 \over d_1} - {n_2 \over d_2} &= {n_1 d_2 - n_2 d_1 \over d_1 d_2}, \\ {n_1 \over d_1} \cdot {n_2 \over d_2} &= {n_1 n_2 \over d_1 d_2}, \\ {n_1 / d_1} \over {n_2 / d_2} &= {n_1 d_2 \over d_1 n_2}, \\ {n_1 \over d_1} &= {n_2 \over d_2} \quad {\rm\ si\ y\ solo\ si\quad} n_1 d_2 = n_2 d_1. \end{aligned}$$

Podemos expresar estas reglas como procedimientos:

(define (add-rat x y) (make-rat (+ (* (numer x) (denom y)) (* (numer y) (denom x))) (* (denom x) (denom y)))) (define (sub-rat x y) (make-rat (- (* (numer x) (denom y)) (* (numer y) (denom x))) (* (denom x) (denom y)))) (define (mul-rat x y) (make-rat (* (numer x) (numer y)) (* (denom x) (denom y)))) (define (div-rat x y) (make-rat (* (numer x) (denom y)) (* (denom x) (numer y))))

(define (equal-rat? x y) (= (* (numer x) (denom y)) (* (numer y) (denom x))))

Ahora tenemos las operaciones sobre números racionales definidas en términos de los procedimientos selectores y constructores `numer`, `denom` y `make/rat`. Pero aún no los hemos definido. Lo que necesitamos es alguna forma de unir un numerador y un denominador para formar un número racional.

Pares

Para permitirnos implementar el nivel concreto de nuestra abstracción de datos, nuestro lenguaje proporciona una estructura compuesta llamada un *par*, que se puede construir con el procedimiento primitivo `cons`. Este procedimiento toma dos argumentos y devuelve un objeto de datos compuesto que contiene los dos argumentos como partes. Dado un par, podemos extraer las partes usando los procedimientos primitivos `car` y `cdr`.[2]

Así, podemos usar `cons`, `car` y `cdr` de la siguiente manera:

(define x (cons 1 2)) (car x) *1* (cdr x) *2*

Observe que un par es un objeto de datos que puede recibir un nombre y ser manipulado, al igual que un objeto de datos primitivo. Además, `cons` se puede usar para formar pares cuyos elementos son pares, y así sucesivamente:

(define x (cons 1 2)) (define y (cons 3 4)) (define z (cons x y)) (car (car z)) *1* (car (cdr z)) *3*

En Section 2.2 veremos cómo esta capacidad de combinar pares significa que los pares se pueden usar como bloques de construcción de propósito general para crear todo tipo de estructuras de datos complejas. El único primitivo de datos compuestos *par*, implementado por los procedimientos `cons`, `car` y `cdr`, es el único pegamento que necesitamos. Los objetos de datos construidos a partir de pares se llaman datos *estructurados en lista*.

Representación de números racionales

Los pares ofrecen una forma natural de completar el sistema de números racionales. Simplemente represente un número racional como un par de dos enteros: un numerador y un denominador. Entonces `make/rat`, `numer` y `denom` se implementan fácilmente de la siguiente manera:[3]

(define (make-rat n d) (cons n d)) (define (numer x) (car x)) (define (denom x) (cdr x))

Además, para mostrar los resultados de nuestros cálculos, podemos imprimir números racionales imprimiendo el numerador, una barra y el denominador:[4]

(define (print-rat x) (newline) (display (numer x)) (display "/") (display (denom x)))

Ahora podemos probar nuestros procedimientos de números racionales:

(define one-half (make-rat 1 2)) (print-rat one-half) *1/2* (define one-third (make-rat 1 3)) (print-rat (add-rat one-half one-third)) *5/6* (print-rat (mul-rat one-half one-third)) *1/6* (print-rat (add-rat one-third one-third)) *6/9*

Como muestra el último ejemplo, nuestra implementación de números racionales no reduce los números racionales a su mínima expresión. Podemos remediar esto cambiando `make/rat`. Si tenemos un procedimiento `gcd` como

el de Section 1.2.5 que produce el máximo común divisor de dos enteros, podemos usar `gcd` para reducir el numerador y el denominador a su mínima expresión antes de construir el par:

```
(define (make-rat n d) (let ((g (gcd n d))) (cons (/ n g) (/ d g))))
```

Ahora tenemos

```
(print-rat (add-rat one-third one-third))  2/3
```

como se deseaba. Esta modificación se logró cambiando el constructor `make/rat` sin cambiar ninguno de los procedimientos (como `add/rat` y `mul/rat`) que implementan las operaciones reales.

> **Ejercicio 2.1:** Defina una versión mejor de `make/rat` que maneje argumentos tanto positivos como negativos. `make/rat` debe normalizar el signo de modo que si el número racional es positivo, tanto el numerador como el denominador sean positivos, y si el número racional es negativo, solo el numerador sea negativo.

Barreras de Abstracción

Antes de continuar con más ejemplos de datos compuestos y abstracción de datos, consideremos algunos de los problemas planteados por el ejemplo de los números racionales. Definimos las operaciones de números racionales en términos de un constructor `make/rat` y los selectores `numer` y `denom`. En general, la idea subyacente de la abstracción de datos es identificar para cada tipo de objeto de datos un conjunto básico de operaciones en términos de las cuales se expresarán todas las manipulaciones de objetos de datos de ese tipo, y luego usar solo esas operaciones para manipular los datos.

> **Figura 2.1:** Barreras de abstracción de datos en el paquete de números racionales.

Podemos visualizar la estructura del sistema de números racionales como se muestra en la Figura 2.1. Las líneas horizontales representan *barreras de abstracción* que aíslan diferentes "niveles" del sistema. En cada nivel, la barrera separa los programas (arriba) que usan la abstracción de datos de los programas (abajo) que implementan la abstracción de datos. Los programas que usan números racionales los manipulan únicamente en términos de los procedimientos proporcionados "para uso público" por el paquete de números racionales: `add/rat`, `sub/rat`, `mul/rat`, `div/rat` y `equal/rat?`. Estos, a su

vez, se implementan únicamente en términos del constructor y los selectores `make/rat`, `numer` y `denom`, que a su vez se implementan en términos de pares. Los detalles de cómo se implementan los pares son irrelevantes para el resto del paquete de números racionales, siempre que los pares se puedan manipular mediante el uso de `cons`, `car` y `cdr`. En efecto, los procedimientos en cada nivel son las interfaces que definen las barreras de abstracción y conectan los diferentes niveles.

Esta simple idea tiene muchas ventajas. Una ventaja es que hace que los programas sean mucho más fáciles de mantener y modificar. Cualquier estructura de datos compleja puede representarse de varias maneras con las estructuras de datos primitivas proporcionadas por un lenguaje de programación. Por supuesto, la elección de la representación influye en los programas que operan sobre ella; por lo tanto, si la representación se cambiara en un momento posterior, todos esos programas podrían tener que modificarse en consecuencia. Esta tarea podría llevar mucho tiempo y ser costosa en el caso de programas grandes, a menos que la dependencia de la representación se confinara por diseño a muy pocos módulos de programa.

Por ejemplo, una forma alternativa de abordar el problema de reducir los números racionales a su mínima expresión es realizar la reducción cada vez que accedemos a las partes de un número racional, en lugar de cuando lo construimos. Esto conduce a diferentes procedimientos de constructor y selector:

(define (make-rat n d) (cons n d)) (define (numer x) (let ((g (gcd (car x) (cdr x)))) (/ (car x) g))) (define (denom x) (let ((g (gcd (car x) (cdr x)))) (/ (cdr x) g)))

La diferencia entre esta implementación y la anterior radica en cuándo calculamos el `gcd`. Si en nuestro uso típico de números racionales accedemos a los numeradores y denominadores de los mismos números racionales muchas veces, sería preferible calcular el `gcd` cuando se construyen los números racionales. Si no, puede ser mejor esperar hasta el momento del acceso para calcular el `gcd`. En cualquier caso, cuando cambiamos de una representación a la otra, los procedimientos `add/rat`, `sub/rat`, etc., no tienen que modificarse en absoluto.

Restringir la dependencia de la representación a unos pocos procedimientos de interfaz nos ayuda a diseñar programas, así como a modificarlos, porque nos permite mantener la flexibilidad para considerar implementaciones alternativas. Para continuar con nuestro simple ejemplo, supongamos que estamos diseñando un paquete de números racionales y no podemos decidir inicialmente si realizar el `gcd` en el momento de la construcción o en el

momento de la selección. La metodología de abstracción de datos nos da una forma de posponer esa decisión sin perder la capacidad de progresar en el resto del sistema.

Ejercicio 2.2: Considere el problema de representar segmentos de línea en un plano. Cada segmento se representa como un par de puntos: un punto de inicio y un punto final. Defina un constructor `make/segment` y los selectores `start/segment` y `end/segment` que definen la representación de los segmentos en términos de puntos. Además, un punto puede representarse como un par de números: la coordenada *x* y la coordenada *y*. En consecuencia, especifique un constructor `make/point` y los selectores `x/point` y `y/point` que definen esta representación. Finalmente, usando sus selectores y constructores, defina un procedimiento `midpoint/segment` que tome un segmento de línea como argumento y devuelva su punto medio (el punto cuyas coordenadas son el promedio de las coordenadas de los puntos finales). Para probar sus procedimientos, necesitará una forma de imprimir puntos:

(define (print-point p) (newline) (display "(") (display (x-point p)) (display ",") (display (y-point p)) (display ")"))

Ejercicio 2.3: Implemente una representación para rectángulos en un plano. (Pista: puede que desee utilizar el Ejercicio 2.2.) En términos de sus constructores y selectores, cree procedimientos que calculen el perímetro y el área de un rectángulo dado. Ahora implemente una representación diferente para los rectángulos. ¿Puede diseñar su sistema con barreras de abstracción adecuadas, de modo que los mismos procedimientos de perímetro y área funcionen con cualquiera de las representaciones?

¿Qué se entiende por Datos?

Comenzamos la implementación de los números racionales en Section 2.1.1 implementando las operaciones de números racionales `add/rat`, `sub/rat`, etc. en términos de tres procedimientos no especificados: `make/rat`, `numer` y `denom`. En ese momento, podíamos pensar en las operaciones como definidas en términos de objetos de datos —numeradores, denominadores y números racionales— cuyo comportamiento estaba especificado por los tres últimos procedimientos.

Pero, ¿qué se entiende exactamente por *datos*? No es suficiente decir "lo que sea que se implemente mediante los selectores y constructores dados". Claramente, no cualquier conjunto arbitrario de tres procedimientos puede servir como una base apropiada para la implementación de números

racionales. Necesitamos garantizar que, si construimos un número racional `x` a partir de un par de enteros `n` y `d`, entonces extraer el `numer` y el `denom` de `x` y dividirlos debería producir el mismo resultado que dividir `n` por `d`. En otras palabras, `make/rat`, `numer` y `denom` deben satisfacer la condición de que, para cualquier entero `n` y cualquier entero no nulo `d`, si `x` es `(make/rat n d)`, entonces

$${\hbox{\tt(numer x)} \over \hbox{\tt(denom x)}} = {{\tt n} \over {\tt d}}\,.$$

De hecho, esta es la única condición que `make/rat`, `numer` y `denom` deben cumplir para formar una base adecuada para una representación de números racionales. En general, podemos pensar en los datos como definidos por alguna colección de selectores y constructores, junto con condiciones especificadas que estos procedimientos deben cumplir para ser una representación válida.[5]

Este punto de vista puede servir para definir no solo objetos de datos de "alto nivel", como los números racionales, sino también objetos de nivel inferior. Considere la noción de un par, que usamos para definir nuestros números racionales. Nunca dijimos realmente qué era un par, solo que el lenguaje proporcionaba los procedimientos `cons`, `car` y `cdr` para operar sobre pares. Pero lo único que necesitamos saber sobre estas tres operaciones es que si unimos dos objetos usando `cons`, podemos recuperar los objetos usando `car` y `cdr`. Es decir, las operaciones satisfacen la condición de que, para cualquier objeto `x` e `y`, si `z` es `(cons x y)`, entonces `(car z)` es `x` y `(cdr z)` es `y`. De hecho, mencionamos que estos tres procedimientos se incluyen como primitivas en nuestro lenguaje. Sin embargo, cualquier triple de procedimientos que satisfaga la condición anterior puede usarse como base para implementar pares. Este punto se ilustra sorprendentemente por el hecho de que podríamos implementar `cons`, `car` y `cdr` sin usar ninguna estructura de datos, sino solo usando procedimientos. Aquí están las definiciones:

(define (cons x y) (define (dispatch m) (cond ((= m 0) x) ((= m 1) y) (else (error "Argument not 0 or 1: CONS" m)))) dispatch) (define (car z) (z 0)) (define (cdr z) (z 1))

Este uso de procedimientos no corresponde a nada parecido a nuestra noción intuitiva de lo que deberían ser los datos. Sin embargo, todo lo que necesitamos hacer para demostrar que esta es una forma válida de representar pares es verificar que estos procedimientos satisfacen la condición dada anteriormente.

El punto sutil a tener en cuenta es que el valor devuelto por `(cons x y)` es un procedimiento, a saber, el procedimiento definido internamente `dispatch`, que toma un argumento y devuelve `x` o `y` dependiendo de si el argumento es 0 o 1. Correspondientemente, `(car z)` se define para aplicar `z` a 0. Por lo tanto, si `z` es el procedimiento formado por `(cons x y)`, entonces `z` aplicado a 0 producirá `x`. Así, hemos demostrado que `(car (cons x y))` produce `x`, como se deseaba. De manera similar, `(cdr (cons x y))` aplica el procedimiento devuelto por `(cons x y)` a 1, que devuelve `y`. Por lo tanto, esta implementación procedimental de pares es una implementación válida, y si accedemos a los pares usando solo `cons`, `car` y `cdr`, no podemos distinguir esta implementación de una que use estructuras de datos "reales".

El propósito de exhibir la representación procedimental de los pares no es que nuestro lenguaje funcione de esta manera (Scheme, y los sistemas Lisp en general, implementan los pares directamente, por razones de eficiencia) sino que podría funcionar de esta manera. La representación procedimental, aunque oscura, es una forma perfectamente adecuada de representar pares, ya que cumple las únicas condiciones que los pares necesitan cumplir. Este ejemplo también demuestra que la capacidad de manipular procedimientos como objetos proporciona automáticamente la capacidad de representar datos compuestos. Esto puede parecer una curiosidad ahora, pero las representaciones procedimentales de datos jugarán un papel central en nuestro repertorio de programación. Este estilo de programación a menudo se llama *paso de mensajes*, y lo usaremos como una herramienta básica en el Capítulo 3 cuando abordemos los problemas de modelado y simulación.

Ejercicio 2.4: Aquí hay una representación procedimental alternativa de pares. Para esta representación, verifique que `(car (cons x y))` produce `x` para cualquier objeto `x` e `y`.

(define (cons x y) (lambda (m) (m x y))) (define (car z) (z (lambda (p q) p)))

¿Cuál es la definición correspondiente de `cdr`? (Pista: Para verificar que esto funciona, utilice el modelo de sustitución de Section 1.1.5).

Ejercicio 2.5: Muestre que podemos representar pares de enteros no negativos usando solo números y operaciones aritméticas si representamos el par *a* y *b* como el entero que es el producto 2^a3^b. Dé las definiciones correspondientes de los procedimientos `cons`, `car` y `cdr`.

Ejercicio 2.6: En caso de que representar pares como procedimientos no fuera lo suficientemente alucinante, considere que, en un lenguaje que puede manipular procedimientos, podemos prescindir de los números (al

menos en lo que respecta a los enteros no negativos) implementando 0 y la operación de sumar 1 como

(define zero (lambda (f) (lambda (x) x))) (define (add-1 n) (lambda (f) (lambda (x) (f ((n f) x)))))

Esta representación se conoce como *numerales de Church*, en honor a su inventor, Alonzo Church, el lógico que inventó el cálculo λ.

Defina `one` y `two` directamente (no en términos de `zero` y `add/1`). (Pista: use la sustitución para evaluar `(add/1 zero)`). Dé una definición directa del procedimiento de suma `+` (no en términos de la aplicación repetida de `add/1`).

Ejercicio Extendido: Aritmética de Intervalos

Alyssa P. Hacker está diseñando un sistema para ayudar a las personas a resolver problemas de ingeniería. Una característica que quiere proporcionar en su sistema es la capacidad de manipular cantidades inexactas (como parámetros medidos de dispositivos físicos) con una precisión conocida, de modo que cuando se realizan cálculos con tales cantidades aproximadas, los resultados sean números de precisión conocida.

Los ingenieros eléctricos usarán el sistema de Alyssa para calcular cantidades eléctricas. A veces es necesario que calculen el valor de una resistencia equivalente en paralelo R_p de dos resistores R_1, R_2 usando la fórmula

$$R_p = {1 \over 1 / R_1 + 1 / R_2}.$$

Los valores de resistencia generalmente solo se conocen hasta cierta tolerancia garantizada por el fabricante del resistor. Por ejemplo, si compra un resistor etiquetado como "6.8 ohmios con 10% de tolerancia", solo puede estar seguro de que el resistor tiene una resistencia entre $6.8 - 0.68 = 6.12$ y $6.8 + 0.68 = 7.48$ ohmios. Por lo tanto, si tiene un resistor de 6.8 ohmios con 10% de tolerancia en paralelo con un resistor de 4.7 ohmios con 5% de tolerancia, la resistencia de la combinación puede variar desde aproximadamente 2.58 ohmios (si los dos resistores están en los límites inferiores) hasta aproximadamente 2.97 ohmios (si los dos resistores están en los límites superiores).

La idea de Alyssa es implementar la "aritmética de intervalos" como un conjunto de operaciones aritméticas para combinar "intervalos" (objetos que representan el rango de valores posibles de una cantidad inexacta). El resultado de sumar, restar, multiplicar o dividir dos intervalos es en sí mismo

un intervalo, que representa el rango del resultado.

Alyssa postula la existencia de un objeto abstracto llamado un "intervalo" que tiene dos puntos finales: un límite inferior y un límite superior. También presume que, dados los puntos finales de un intervalo, puede construir el intervalo usando el constructor de datos `make/interval`. Alyssa primero escribe un procedimiento para sumar dos intervalos. Razona que el valor mínimo que podría tener la suma es la suma de los dos límites inferiores y el valor máximo que podría tener es la suma de los dos límites superiores:

```
(define (add-interval x y) (make-interval (+ (lower-bound x) (lower-bound y)) (+ (upper-bound x) (upper-bound y))))
```

Alyssa también calcula el producto de dos intervalos encontrando el mínimo y el máximo de los productos de los límites y usándolos como los límites del intervalo resultante. (`min` y `max` son primitivas que encuentran el mínimo o el máximo de cualquier número de argumentos).

```
(define (mul-interval x y) (let ((p1 (* (lower-bound x) (lower-bound y))) (p2 (* (lower-bound x) (upper-bound y))) (p3 (* (upper-bound x) (lower-bound y))) (p4 (* (upper-bound x) (upper-bound y)))) (make-interval (min p1 p2 p3 p4) (max p1 p2 p3 p4))))
```

Para dividir dos intervalos, Alyssa multiplica el primero por el recíproco del segundo. Tenga en cuenta que los límites del intervalo recíproco son el recíproco del límite superior y el recíproco del límite inferior, en ese orden.

```
(define (div-interval x y) (mul-interval x (make-interval (/ 1.0 (upper-bound y)) (/ 1.0 (lower-bound y)))))
```

> **Ejercicio 2.7:** El programa de Alyssa está incompleto porque no ha especificado la implementación de la abstracción de intervalo. Aquí hay una definición del constructor de intervalo:
>
> ```
> (define (make-interval a b) (cons a b))
> ```
>
> Defina los selectores `upper/bound` y `lower/bound` para completar la implementación.
>
> **Ejercicio 2.8:** Usando un razonamiento análogo al de Alyssa, describa cómo se puede calcular la diferencia de dos intervalos. Defina un procedimiento de resta correspondiente, llamado `sub/interval`.
>
> **Ejercicio 2.9:** La *anchura* de un intervalo es la mitad de la diferencia entre sus límites superior e inferior. La anchura es una medida de la

incertidumbre del número especificado por el intervalo. Para algunas operaciones aritméticas, la anchura del resultado de combinar dos intervalos es una función solo de las anchuras de los intervalos de los argumentos, mientras que para otras, la anchura de la combinación no es una función de las anchuras de los intervalos de los argumentos. Muestre que la anchura de la suma (o diferencia) de dos intervalos es una función solo de las anchuras de los intervalos que se están sumando (o restando). Dé ejemplos para mostrar que esto no es cierto para la multiplicación o la división.

Ejercicio 2.10: Ben Bitdiddle, un programador experto en sistemas, mira por encima del hombro de Alyssa y comenta que no está claro qué significa dividir por un intervalo que abarca el cero. Modifique el código de Alyssa para verificar esta condición y señalar un error si ocurre.

Ejercicio 2.11: De paso, Ben también comenta crípticamente: "Al probar los signos de los puntos finales de los intervalos, es posible dividir `mul/interval` en nueve casos, de los cuales solo uno requiere más de dos multiplicaciones". Reescriba este procedimiento usando la sugerencia de Ben.

Después de depurar su programa, Alyssa se lo muestra a un usuario potencial, quien se queja de que su programa resuelve el problema equivocado. Él quiere un programa que pueda tratar con números representados como un valor central y una tolerancia aditiva; por ejemplo, quiere trabajar con intervalos como 3.5 ± 0.15 en lugar de [3.35, 3.65]. Alyssa regresa a su escritorio y soluciona este problema proporcionando un constructor alternativo y selectores alternativos:

(define (make-center-width c w) (make-interval (- c w) (+ c w))) (define (center i) (/ (+ (lower-bound i) (upper-bound i)) 2)) (define (width i) (/ (- (upper-bound i) (lower-bound i)) 2))

Desafortunadamente, la mayoría de los usuarios de Alyssa son ingenieros. Las situaciones de ingeniería reales generalmente involucran mediciones con solo una pequeña incertidumbre, medida como la relación entre la anchura del intervalo y el punto medio del intervalo. Los ingenieros generalmente especifican tolerancias porcentuales en los parámetros de los dispositivos, como en las especificaciones de resistores dadas anteriormente.

Ejercicio 2.12: Defina un constructor `make/center/percent` que tome un centro y una tolerancia porcentual y produzca el intervalo deseado. También debe definir un selector `percent` que produzca la tolerancia

porcentual para un intervalo dado. El selector `center` es el mismo que el que se muestra arriba.

Ejercicio 2.13: Muestre que bajo la suposición de pequeñas tolerancias porcentuales hay una fórmula simple para la tolerancia porcentual aproximada del producto de dos intervalos en términos de las tolerancias de los factores. Puede simplificar el problema asumiendo que todos los números son positivos.

Después de un trabajo considerable, Alyssa P. Hacker entrega su sistema terminado. Varios años después, después de que se haya olvidado por completo de él, recibe una llamada frenética de un usuario furioso, Lem E. Tweakit. Parece que Lem ha notado que la fórmula para resistores en paralelo se puede escribir de dos maneras algebraicamente equivalentes:

$$R_1 R_2 \over R_1 + R_2$$

y

$${1 \over 1 / R_1 + 1 / R_2}.$$

Ha escrito los siguientes dos programas, cada uno de los cuales calcula la fórmula de los resistores en paralelo de manera diferente:

(define (par1 r1 r2) (div-interval (mul-interval r1 r2) (add-interval r1 r2)))

(define (par2 r1 r2) (let ((one (make-interval 1 1))) (div-interval one (add-interval (div-interval one r1) (div-interval one r2)))))

Lem se queja de que el programa de Alyssa da respuestas diferentes para las dos formas de calcular. Esta es una queja seria.

Ejercicio 2.14: Demuestre que Lem tiene razón. Investigue el comportamiento del sistema en una variedad de expresiones aritméticas. Cree algunos intervalos *A* y *B*, y úselos para calcular las expresiones *A/A* y *A/B*. Obtendrá la mayor comprensión usando intervalos cuya anchura sea un pequeño porcentaje del valor central. Examine los resultados del cálculo en forma de centro-porcentaje (ver Ejercicio 2.12).

Ejercicio 2.15: Eva Lu Ator, otra usuaria, también ha notado los diferentes intervalos calculados por expresiones diferentes pero algebraicamente equivalentes. Dice que una fórmula para calcular con intervalos usando el sistema de Alyssa producirá límites de error más ajustados si se puede escribir de tal forma que ninguna variable que

represente un número incierto se repita. Por lo tanto, dice, `par2` es un programa "mejor" para resistencias en paralelo que `par1`. ¿Tiene razón? ¿Por qué?

Ejercicio 2.16: Explique, en general, por qué expresiones algebraicas equivalentes pueden llevar a respuestas diferentes. ¿Puede idear un paquete de aritmética de intervalos que no tenga esta deficiencia, o es esta tarea imposible? (Advertencia: este problema es muy difícil).

Datos Jerárquicos y la Propiedad de Clausura

Como hemos visto, los pares proporcionan un "pegamento" primitivo que podemos usar para construir objetos de datos compuestos. La Figura 2.2 muestra una forma estándar de visualizar un par, en este caso, el par formado por `(cons 1 2)`. En esta representación, que se llama *notación de caja y puntero*, cada objeto se muestra como un *puntero* a una caja. La caja para un objeto primitivo contiene una representación del objeto. Por ejemplo, la caja para un número contiene un numeral. La caja para un par es en realidad una caja doble, la parte izquierda contiene (un puntero a) el `car` del par y la parte derecha contiene el `cdr`.

Ya hemos visto que `cons` se puede usar para combinar no solo números, sino también pares. (Hizo uso de este hecho, o debería haberlo hecho, al hacer los Ejercicio 2.2 y Ejercicio 2.3). Como consecuencia, los pares proporcionan un bloque de construcción universal a partir del cual podemos construir todo tipo de estructuras de datos. La Figura 2.3 muestra dos formas de usar pares para combinar los números 1, 2, 3 y 4.

Figura 2.2: Representación de caja y puntero de `(cons 1 2)`.

Figura 2.3: Dos formas de combinar 1, 2, 3 y 4 usando pares.

La capacidad de crear pares cuyos elementos son pares es la esencia de la importancia de la estructura de lista como herramienta de representación. Nos referimos a esta capacidad como la *propiedad de clausura* de `cons`. En general, una operación para combinar objetos de datos satisface la propiedad de clausura si los resultados de combinar cosas con esa operación pueden a su vez combinarse usando la misma operación.[6] La clausura es la clave del poder en cualquier medio de combinación porque nos permite crear

estructuras *jerárquicas*, estructuras compuestas de partes, que a su vez están compuestas de partes, y así sucesivamente.

Desde el comienzo del Capítulo 1, hemos hecho un uso esencial de la clausura al tratar con procedimientos, porque todos los programas, excepto los más simples, dependen del hecho de que los elementos de una combinación pueden ser a su vez combinaciones. En esta sección, abordamos las consecuencias de la clausura para los datos compuestos. Describimos algunas técnicas convencionales para usar pares para representar secuencias y árboles, y exhibimos un lenguaje de gráficos que ilustra la clausura de una manera vívida.[7]

Representando Secuencias

Una de las estructuras útiles que podemos construir con pares es una *secuencia*, una colección ordenada de objetos de datos. Hay, por supuesto, muchas maneras de representar secuencias en términos de pares. Una representación particularmente sencilla se ilustra en la Figura 2.4, donde la secuencia 1, 2, 3, 4 se representa como una cadena de pares. El `car` de cada par es el elemento correspondiente en la cadena, y el `cdr` del par es el siguiente par en la cadena. El `cdr` del par final señala el final de la secuencia al apuntar a un valor distinguido que no es un par, representado en los diagramas de caja y puntero como una línea diagonal y en los programas como el valor de la variable `nil`. La secuencia completa se construye mediante operaciones `cons` anidadas:

(cons 1 (cons 2 (cons 3 (cons 4 nil))))

Figura 2.4: La secuencia 1, 2, 3, 4 representada como una cadena de pares.

Tal secuencia de pares, formada por `cons` anidados, se llama una *lista*, y Scheme proporciona una primitiva llamada `list` para ayudar en la construcción de listas.[8] La secuencia anterior podría ser producida por `(list 1 2 3 4)`. En general,

(list < *a* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$ < *a* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$ … < *a* $\color{SchemeDark}_{\hbox{\ttfamily\itshape\scriptsize n}}\rangle$)

es equivalente a

(cons ⟨ *a* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$ (cons ⟨ *a* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$ (cons … (cons ⟨ *a* $\color{SchemeDark}_{\hbox{\ttfamily\itshape \scriptsize n}}\rangle$ nil) …)))

Los sistemas Lisp imprimen convencionalmente las listas imprimiendo la secuencia de elementos, encerrada entre paréntesis. Por lo tanto, el objeto de datos en la Figura 2.4 se imprime como `(1 2 3 4)`:

(define one-through-four (list 1 2 3 4)) one-through-four *(1 2 3 4)*

Tenga cuidado de no confundir la expresión `(list 1 2 3 4)` con la lista `(1 2 3 4)`, que es el resultado obtenido cuando se evalúa la expresión. Intentar evaluar la expresión `(1 2 3 4)` señalará un error cuando el intérprete intente aplicar el procedimiento `1` a los argumentos `2`, `3` y `4`.

Podemos pensar en `car` como la selección del primer elemento de la lista, y en `cdr` como la selección de la sublista que consiste en todo menos el primer elemento. Las aplicaciones anidadas de `car` y `cdr` se pueden usar para extraer el segundo, tercero y subsiguientes elementos de la lista.[9] El constructor `cons` crea una lista como la original, pero con un elemento adicional al principio.

(car one-through-four) *1* (cdr one-through-four) *(2 3 4)* (car (cdr one-through-four)) *2* (cons 10 one-through-four) *(10 1 2 3 4)* (cons 5 one-through-four) *(5 1 2 3 4)*

El valor de `nil`, utilizado para terminar la cadena de pares, puede considerarse como una secuencia sin elementos, la *lista vacía*. La palabra *nil* es una contracción de la palabra latina *nihil*, que significa "nada".[10]

Operaciones de lista

El uso de pares para representar secuencias de elementos como listas va acompañado de técnicas de programación convencionales para manipular listas recorriéndolas sucesivamente con "‘`cdr`’". Por ejemplo, el procedimiento `list/ref` toma como argumentos una lista y un número n y devuelve el n^{th} elemento de la lista. Se acostumbra a numerar los elementos de la lista comenzando con 0. El método para calcular `list/ref` es el siguiente:

- Para $n = 0$, `list/ref` debe devolver el `car` de la lista.
- De lo contrario, `list/ref` debe devolver el $(n - 1)$-ésimo elemento del `cdr` de la lista.

(define (list-ref items n) (if (= n 0) (car items) (list-ref (cdr items) (- n 1)))) (define squares (list 1 4 9 16 25)) (list-ref squares 3) *16*

A menudo recorremos toda la lista con `cdr`. Para ayudar en esto, Scheme incluye un predicado primitivo `null?`, que prueba si su argumento es la lista vacía. El procedimiento `length`, que devuelve el número de elementos en una lista, ilustra este patrón típico de uso:

(define (length items) (if (null? items) 0 (+ 1 (length (cdr items))))) (define odds (list 1 3 5 7)) (length odds) *4*

El procedimiento `length` implementa un plan recursivo simple. El paso de reducción es:

- La `longitud` de cualquier lista es 1 más la `longitud` del `cdr` de la lista.

Esto se aplica sucesivamente hasta que llegamos al caso base:

- La `longitud` de la lista vacía es 0.

También podríamos calcular `length` en un estilo iterativo:

(define (length items) (define (length-iter a count) (if (null? a) count (length-iter (cdr a) (+ 1 count)))) (length-iter items 0))

Otra técnica de programación convencional es "construir con `cons`" una lista de respuestas mientras se recorre una lista con `cdr`, como en el procedimiento `append`, que toma dos listas como argumentos y combina sus elementos para crear una nueva lista:

(append squares odds) *(1 4 9 16 25 1 3 5 7)* (append odds squares) *(1 3 5 7 1 4 9 16 25)*

`append` también se implementa usando un plan recursivo. Para `append` las listas `list1` y `list2`, haga lo siguiente:

- Si `list1` es la lista vacía, entonces el resultado es simplemente `list2`.
- De lo contrario, `append` el `cdr` de `list1` y `list2`, y `cons` el `car` de `list1` al resultado:

(define (append list1 list2) (if (null? list1) list2 (cons (car list1) (append (cdr list1) list2))))

Ejercicio 2.17: Defina un procedimiento `last/pair` que devuelva la lista que contiene solo el último elemento de una lista dada (no vacía):

(last-pair (list 23 72 149 34)) *(34)*

Ejercicio 2.18: Defina un procedimiento `reverse` que tome una lista como argumento y devuelva una lista de los mismos elementos en orden inverso:

(reverse (list 1 4 9 16 25)) *(25 16 9 4 1)*

Ejercicio 2.19: Considere el programa de conteo de cambio de Section 1.2.2. Sería bueno poder cambiar fácilmente la moneda utilizada por el programa, de modo que pudiéramos calcular el número de formas de cambiar una libra esterlina, por ejemplo. Tal como está escrito el programa, el conocimiento de la moneda se distribuye en parte en el procedimiento `first/denomination` y en parte en el procedimiento `count/change` (que sabe que hay cinco tipos de monedas de EE. UU.). Sería mejor poder proporcionar una lista de monedas para usar en el cambio.

Queremos reescribir el procedimiento `cc` para que su segundo argumento sea una lista de los valores de las monedas a usar en lugar de un entero que especifique qué monedas usar. Podríamos entonces tener listas que definieran cada tipo de moneda:

(define us-coins (list 50 25 10 5 1)) (define uk-coins (list 100 50 20 10 5 2 1 0.5))

Podríamos entonces llamar a `cc` de la siguiente manera:

(cc 100 us-coins) *292*

Para hacer esto será necesario cambiar un poco el programa `cc`. Todavía tendrá la misma forma, pero accederá a su segundo argumento de manera diferente, de la siguiente manera:

(define (cc amount coin-values) (cond ((= amount 0) 1) ((or (< amount 0) (no-more? coin-values)) 0) (else (+ (cc amount (except-first-denomination coin-values)) (cc (- amount (first-denomination coin-values)) coin-values)))))

Defina los procedimientos `first/denomination`, `except/first/denomination` y `no/more?` en términos de operaciones primitivas sobre estructuras de lista. ¿Afecta el orden de la lista `coin/values` a la respuesta producida por `cc`? ¿Por qué o por qué no?

Ejercicio 2.20: Los procedimientos `+`, `*` y `list` toman un número

arbitrario de argumentos. Una forma de definir tales procedimientos es usar `define` con la *notación de cola punteada*. En una definición de procedimiento, una lista de parámetros que tiene un punto antes del último nombre del parámetro indica que, cuando se llama al procedimiento, los parámetros iniciales (si los hay) tendrán como valores los argumentos iniciales, como de costumbre, pero el valor del parámetro final será una *lista* de cualquier argumento restante. Por ejemplo, dada la definición

(define (f x y . z) ⟨ *cuerpo* ⟩)

el procedimiento `f` puede ser llamado con dos o más argumentos. Si evaluamos

(f 1 2 3 4 5 6)

entonces en el cuerpo de `f`, `x` será 1, `y` será 2, y `z` será la lista `(3 4 5 6)`. Dada la definición

(define (g . w) ⟨ *cuerpo* ⟩)

el procedimiento `g` puede ser llamado con cero o más argumentos. Si evaluamos

(g 1 2 3 4 5 6)

entonces en el cuerpo de `g`, `w` será la lista `(1 2 3 4 5 6)`.[11]

Use esta notación para escribir un procedimiento `same/parity` que tome uno o más enteros y devuelva una lista de todos los argumentos que tienen la misma paridad par-impar que el primer argumento. Por ejemplo,

(same-parity 1 2 3 4 5 6 7) *(1 3 5 7)* (same-parity 2 3 4 5 6 7) *(2 4 6)*

Mapeo sobre listas

Una operación extremadamente útil es aplicar alguna transformación a cada elemento de una lista y generar la lista de resultados. Por ejemplo, el siguiente procedimiento escala cada número en una lista por un factor dado:

(define (scale-list items factor) (if (null? items) nil (cons (* (car items) factor) (scale-list (cdr items) factor)))) (scale-list (list 1 2 3 4 5) 10) *(10 20 30 40 50)*

Podemos abstraer esta idea general y capturarla como un patrón común expresado como un procedimiento de orden superior, tal como en Section 1.3. El procedimiento de orden superior aquí se llama `map`. `map` toma como argumentos un procedimiento de un argumento y una lista, y devuelve una lista de los resultados producidos al aplicar el procedimiento a cada elemento de la lista:[12]

(define (map proc items) (if (null? items) nil (cons (proc (car items)) (map proc (cdr items))))) (map abs (list -10 2.5 -11.6 17)) *(10 2.5 11.6 17)* (map (lambda (x) (* x x)) (list 1 2 3 4)) *(1 4 9 16)*

Ahora podemos dar una nueva definición de `scale/list` en términos de `map`:

(define (scale-list items factor) (map (lambda (x) (* x factor)) items))

`map` es una construcción importante, no solo porque captura un patrón común, sino porque establece un nivel superior de abstracción al tratar con listas. En la definición original de `scale/list`, la estructura recursiva del programa llama la atención sobre el procesamiento elemento por elemento de la lista. Definir `scale/list` en términos de `map` suprime ese nivel de detalle y enfatiza que la escala transforma una lista de elementos en una lista de resultados. La diferencia entre las dos definiciones no es que el ordenador esté realizando un proceso diferente (no lo está), sino que pensamos en el proceso de manera diferente. En efecto, `map` ayuda a establecer una barrera de abstracción que aísla la implementación de los procedimientos que transforman listas de los detalles de cómo se extraen y combinan los elementos de la lista. Al igual que las barreras que se muestran en la Figura 2.1, esta abstracción nos da la flexibilidad para cambiar los detalles de bajo nivel de cómo se implementan las secuencias, al tiempo que se preserva el marco conceptual de las operaciones que transforman secuencias en secuencias. Section 2.2.3 amplía este uso de las secuencias como marco para organizar programas.

> **Ejercicio 2.21:** El procedimiento `square/list` toma una lista de números como argumento y devuelve una lista de los cuadrados de esos números.
>
> (square-list (list 1 2 3 4)) *(1 4 9 16)*
>
> Aquí hay dos definiciones diferentes de `square/list`. Complete ambas rellenando las expresiones que faltan:
>
> (define (square-list items) (if (null? items) nil (cons < ?? > < ?? >))) (define (square-list items) (map < ?? > < ?? >))

Ejercicio 2.22: Louis Reasoner intenta reescribir el primer procedimiento `square/list` del Ejercicio 2.21 para que desarrolle un proceso iterativo:

(define (square-list items) (define (iter things answer) (if (null? things) answer (iter (cdr things) (cons (square (car things)) answer)))) (iter items nil))

Desafortunadamente, definir `square/list` de esta manera produce la lista de respuestas en el orden inverso al deseado. ¿Por qué?

Louis luego intenta corregir su error intercambiando los argumentos de `cons`:

(define (square-list items) (define (iter things answer) (if (null? things) answer (iter (cdr things) (cons answer (square (car things)))))) (iter items nil))

Esto tampoco funciona. Explique.

Ejercicio 2.23: El procedimiento `for/each` es similar a `map`. Toma como argumentos un procedimiento y una lista de elementos. Sin embargo, en lugar de formar una lista de los resultados, `for/each` simplemente aplica el procedimiento a cada uno de los elementos a su vez, de izquierda a derecha. Los valores devueltos al aplicar el procedimiento a los elementos no se utilizan en absoluto; `for/each` se utiliza con procedimientos que realizan una acción, como imprimir. Por ejemplo,

(for-each (lambda (x) (newline) (display x)) (list 57 321 88)) *57 321 88*

El valor devuelto por la llamada a `for/each` (no ilustrado anteriormente) puede ser algo arbitrario, como verdadero. Dé una implementación de `for/each`.

Estructuras Jerárquicas

La representación de secuencias en términos de listas se generaliza naturalmente para representar secuencias cuyos elementos pueden ser a su vez secuencias. Por ejemplo, podemos considerar el objeto `((1 2) 3 4)` construido por

(cons (list 1 2) (list 3 4))

como una lista de tres elementos, el primero de los cuales es a su vez una lista, `(1 2)`. De hecho, esto lo sugiere la forma en que el resultado es

impreso por el intérprete. La Figura 2.5 muestra la representación de esta estructura en términos de pares.

Figura 2.5: Estructura formada por `(cons (list 1 2) (list 3 4))`.

Otra forma de pensar en secuencias cuyos elementos son secuencias es como *árboles*. Los elementos de la secuencia son las ramas del árbol, y los elementos que son a su vez secuencias son subárboles. La Figura 2.6 muestra la estructura en la Figura 2.5 vista como un árbol.

Figura 2.6: La estructura de lista en la Figura 2.5 vista como un árbol.

La recursión es una herramienta natural para tratar con estructuras de árbol, ya que a menudo podemos reducir las operaciones en árboles a operaciones en sus ramas, que a su vez se reducen a operaciones en las ramas de las ramas, y así sucesivamente, hasta que llegamos a las hojas del árbol. Como ejemplo, compare el procedimiento `length` de Section 2.2.1 con el procedimiento `count/leaves`, que devuelve el número total de hojas de un árbol:

(define x (cons (list 1 2) (list 3 4))) (length x) *3* (count-leaves x) *4* (list x x) *(((1 2) 3 4) ((1 2) 3 4))* (length (list x x)) *2* (count-leaves (list x x)) *8*

Para implementar `count/leaves`, recuerde el plan recursivo para calcular `length`:

- `length` de una lista `x` es 1 más `length` del `cdr` de `x`.
- `length` de la lista vacía es 0.

`count/leaves` es similar. El valor para la lista vacía es el mismo:

- `count/leaves` de la lista vacía es 0.

Pero en el paso de reducción, donde eliminamos el `car` de la lista, debemos tener en cuenta que el `car` puede ser a su vez un árbol cuyas hojas necesitamos contar. Por lo tanto, el paso de reducción apropiado es

- `count/leaves` de un árbol `x` es `count/leaves` del `car` de `x` más `count/leaves` del `cdr` de `x`.

Finalmente, al tomar `car`s llegamos a hojas reales, por lo que necesitamos

otro caso base:

- `count/leaves` de una hoja es 1.

Para ayudar a escribir procedimientos recursivos en árboles, Scheme proporciona el predicado primitivo `pair?`, que prueba si su argumento es un par. Aquí está el procedimiento completo:[13]

(define (count-leaves x) (cond ((null? x) 0) ((not (pair? x)) 1) (else (+ (count-leaves (car x)) (count-leaves (cdr x))))))

Ejercicio 2.24: Supongamos que evaluamos la expresión `(list 1 (list 2 (list 3 4)))`. Dé el resultado impreso por el intérprete, la estructura de caja y puntero correspondiente, y la interpretación de esto como un árbol (como en la Figura 2.6).

Ejercicio 2.25: Dé combinaciones de `car`s y `cdr`s que seleccionarán el 7 de cada una de las siguientes listas:

(1 3 (5 7) 9) ((7)) (1 (2 (3 (4 (5 (6 7))))))

Ejercicio 2.26: Supongamos que definimos `x` e `y` como dos listas:

(define x (list 1 2 3)) (define y (list 4 5 6))

¿Qué resultado imprime el intérprete en respuesta a la evaluación de cada una de las siguientes expresiones?:

(append x y) (cons x y) (list x y)

Ejercicio 2.27: Modifique su procedimiento `reverse` del Ejercicio 2.18 para producir un procedimiento `deep/reverse` que tome una lista como argumento y devuelva como valor la lista con sus elementos invertidos y con todas las sublistas también invertidas profundamente. Por ejemplo,

(define x (list (list 1 2) (list 3 4))) x *((1 2) (3 4))* (reverse x) *((3 4) (1 2))* (deep-reverse x) *((4 3) (2 1))*

Ejercicio 2.28: Escriba un procedimiento `fringe` que tome como argumento un árbol (representado como una lista) y devuelva una lista cuyos elementos son todas las hojas del árbol dispuestas en orden de izquierda a derecha. Por ejemplo,

(define x (list (list 1 2) (list 3 4))) (fringe x) *(1 2 3 4)* (fringe (list x x)) *(1 2 3 4 1 2 3 4)*

Ejercicio 2.29: Un móvil binario consiste en dos ramas, una rama izquierda y una rama derecha. Cada rama es una varilla de una cierta longitud, de la que cuelga o un peso o otro móvil binario. Podemos representar un móvil binario usando datos compuestos construyéndolo a partir de dos ramas (por ejemplo, usando `list`):

(define (make-mobile left right) (list left right))

Una rama se construye a partir de una `longitud` (que debe ser un número) junto con una `estructura`, que puede ser o un número (representando un peso simple) u otro móvil:

(define (make-branch length structure) (list length structure))

1. Escriba los selectores correspondientes `left/branch` y `right/branch`, que devuelven las ramas de un móvil, y `branch/length` y `branch/structure`, que devuelven los componentes de una rama.

2. Usando sus selectores, defina un procedimiento `total/weight` que devuelva el peso total de un móvil.

3. Se dice que un móvil está *equilibrado* si el par de torsión aplicado por su rama superior izquierda es igual al aplicado por su rama superior derecha (es decir, si la longitud de la varilla izquierda multiplicada por el peso que cuelga de esa varilla es igual al producto correspondiente para el lado derecho) y si cada uno de los submóviles que cuelgan de sus ramas está equilibrado. Diseñe un predicado que pruebe si un móvil binario está equilibrado.

4. Suponga que cambiamos la representación de los móviles para que los constructores sean

   (define (make-mobile left right) (cons left right)) (define (make-branch length structure) (cons length structure))

   ¿Cuánto necesita cambiar sus programas para convertirlos a la nueva representación?

Mapeo sobre árboles

Así como `map` es una abstracción poderosa para tratar con secuencias, `map` junto con la recursión es una abstracción poderosa para tratar con árboles. Por ejemplo, el procedimiento `scale/tree`, análogo a `scale/list` de Section 2.2.1, toma como argumentos un factor numérico y un árbol cuyas hojas son números. Devuelve un árbol de la misma forma, donde cada número se

multiplica por el factor. El plan recursivo para `scale/tree` es similar al de `count/leaves`:

(define (scale-tree tree factor) (cond ((null? tree) nil) ((not (pair? tree)) (* tree factor)) (else (cons (scale-tree (car tree) factor) (scale-tree (cdr tree) factor))))) (scale-tree (list 1 (list 2 (list 3 4) 5) (list 6 7)) 10) *(10 (20 (30 40) 50) (60 70))*

Otra forma de implementar `scale/tree` es considerar el árbol como una secuencia de subárboles y usar `map`. Mapeamos sobre la secuencia, escalando cada subárbol a su vez, y devolvemos la lista de resultados. En el caso base, donde el árbol es una hoja, simplemente multiplicamos por el factor:

(define (scale-tree tree factor) (map (lambda (sub-tree) (if (pair? sub-tree) (scale-tree sub-tree factor) (* sub-tree factor))) tree))

Muchas operaciones de árbol se pueden implementar mediante combinaciones similares de operaciones de secuencia y recursión.

Ejercicio 2.30: Defina un procedimiento `square/tree` análogo al procedimiento `square/list` del Ejercicio 2.21. Es decir, `square/tree` debería comportarse de la siguiente manera:

(square-tree (list 1 (list 2 (list 3 4) 5) (list 6 7))) *(1 (4 (9 16) 25) (36 49))*

Defina `square/tree` tanto directamente (es decir, sin usar ningún procedimiento de orden superior) como también usando `map` y recursión.

Ejercicio 2.31: Abstraiga su respuesta al Ejercicio 2.30 para producir un procedimiento `tree/map` con la propiedad de que `square/tree` podría definirse como

(define (square-tree tree) (tree-map square tree))

Ejercicio 2.32: Podemos representar un conjunto como una lista de elementos distintos, y podemos representar el conjunto de todos los subconjuntos del conjunto como una lista de listas. Por ejemplo, si el conjunto es `(1 2 3)`, entonces el conjunto de todos los subconjuntos es `(() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))`. Complete la siguiente definición de un procedimiento que genera el conjunto de subconjuntos de un conjunto y dé una explicación clara de por qué funciona:

(define (subsets s) (if (null? s) (list nil) (let ((rest (subsets (cdr s)))) (append rest (map < ?? > rest)))))

Secuencias como Interfaces Convencionales

Al trabajar con datos compuestos, hemos enfatizado cómo la abstracción de datos nos permite diseñar programas sin enredarnos en los detalles de las representaciones de datos, y cómo la abstracción nos preserva la flexibilidad para experimentar con representaciones alternativas. En esta sección, introducimos otro poderoso principio de diseño para trabajar con estructuras de datos: el uso de *interfaces convencionales*.

En Section 1.3 vimos cómo las abstracciones de programas, implementadas como procedimientos de orden superior, pueden capturar patrones comunes en programas que tratan con datos numéricos. Nuestra capacidad para formular operaciones análogas para trabajar con datos compuestos depende crucialmente del estilo en que manipulemos nuestras estructuras de datos. Considere, por ejemplo, el siguiente procedimiento, análogo al procedimiento `count/leaves` de Section 2.2.2, que toma un árbol como argumento y calcula la suma de los cuadrados de las hojas que son impares:

(define (sum-odd-squares tree) (cond ((null? tree) 0) ((not (pair? tree)) (if (odd? tree) (square tree) 0)) (else (+ (sum-odd-squares (car tree)) (sum-odd-squares (cdr tree))))))

A primera vista, este procedimiento es muy diferente del siguiente, que construye una lista de todos los números de Fibonacci pares ${\rm Fib}(k)$, donde *k* es menor o igual que un entero dado *n*:

(define (even-fibs n) (define (next k) (if (> k n) nil (let ((f (fib k))) (if (even? f) (cons f (next (+ k 1))) (next (+ k 1)))))) (next 0))

A pesar de que estos dos procedimientos son estructuralmente muy diferentes, una descripción más abstracta de los dos cálculos revela una gran similitud. El primer programa

- enumera las hojas de un árbol;
- las filtra, seleccionando las impares;
- eleva al cuadrado cada una de las seleccionadas; y
- acumula los resultados usando `+`, comenzando con 0.

El segundo programa

- enumera los enteros de 0 a *n*;

- calcula el número de Fibonacci para cada entero;
- los filtra, seleccionando los pares; y
- acumula los resultados usando `cons`, comenzando con la lista vacía.

Figura 2.7: Los planes de flujo de señales para los procedimientos `sum/odd/squares` (arriba) y `even/fibs` (abajo) revelan la similitud entre los dos programas.

Un ingeniero de procesamiento de señales encontraría natural conceptualizar estos procesos en términos de señales que fluyen a través de una cascada de etapas, cada una de las cuales implementa parte del plan del programa, como se muestra en la Figura 2.7. En `sum/odd/squares`, comenzamos con un *enumerador*, que genera una "señal" que consiste en las hojas de un árbol dado. Esta señal pasa a través de un *filtro*, que elimina todos los elementos excepto los impares. La señal resultante pasa a su vez a través de un *mapa*, que es un "transductor" que aplica el procedimiento `square` a cada elemento. La salida del mapa se alimenta luego a un *acumulador*, que combina los elementos usando `+`, comenzando desde un 0 inicial. El plan para `even/fibs` es análogo.

Desafortunadamente, las dos definiciones de procedimiento anteriores no exhiben esta estructura de flujo de señales. Por ejemplo, si examinamos el procedimiento `sum/odd/squares`, encontramos que la enumeración se implementa en parte mediante las pruebas `null?` y `pair?` y en parte mediante la estructura recursiva de árbol del procedimiento. De manera similar, la acumulación se encuentra en parte en las pruebas y en parte en la suma utilizada en la recursión. En general, no hay partes distintas de ninguno de los procedimientos que correspondan a los elementos en la descripción del flujo de señales. Nuestros dos procedimientos descomponen los cálculos de una manera diferente, extendiendo la enumeración sobre el programa y mezclándola con el mapa, el filtro y la acumulación. Si pudiéramos organizar nuestros programas para que la estructura de flujo de señales se manifieste en los procedimientos que escribimos, esto aumentaría la claridad conceptual del código resultante.

Operaciones de Secuencia

La clave para organizar programas de manera que reflejen más claramente la estructura de flujo de señales es concentrarse en las "señales" que fluyen de una etapa a la siguiente en el proceso. Si representamos estas señales como

listas, entonces podemos usar operaciones de lista para implementar el procesamiento en cada una de las etapas. Por ejemplo, podemos implementar las etapas de mapeo de los diagramas de flujo de señales usando el procedimiento `map` de Section 2.2.1:

(map square (list 1 2 3 4 5)) *(1 4 9 16 25)*

Filtrar una secuencia para seleccionar solo aquellos elementos que satisfacen un predicado dado se logra mediante

(define (filter predicate sequence) (cond ((null? sequence) nil) ((predicate (car sequence)) (cons (car sequence) (filter predicate (cdr sequence)))) (else (filter predicate (cdr sequence)))))

Por ejemplo,

(filter odd? (list 1 2 3 4 5)) *(1 3 5)*

Las acumulaciones se pueden implementar mediante

(define (accumulate op initial sequence) (if (null? sequence) initial (op (car sequence) (accumulate op initial (cdr sequence))))) (accumulate + 0 (list 1 2 3 4 5)) *15* (accumulate * 1 (list 1 2 3 4 5)) *120* (accumulate cons nil (list 1 2 3 4 5)) *(1 2 3 4 5)*

Todo lo que queda para implementar los diagramas de flujo de señales es enumerar la secuencia de elementos a procesar. Para `even/fibs`, necesitamos generar la secuencia de enteros en un rango dado, lo que podemos hacer de la siguiente manera:

(define (enumerate-interval low high) (if (> low high) nil (cons low (enumerate-interval (+ low 1) high)))) (enumerate-interval 2 7) *(2 3 4 5 6 7)*

Para enumerar las hojas de un árbol, podemos usar[14]

(define (enumerate-tree tree) (cond ((null? tree) nil) ((not (pair? tree)) (list tree)) (else (append (enumerate-tree (car tree)) (enumerate-tree (cdr tree)))))) (enumerate-tree (list 1 (list 2 (list 3 4)) 5)) *(1 2 3 4 5)*

Ahora podemos reformular `sum/odd/squares` y `even/fibs` como en los diagramas de flujo de señales. Para `sum/odd/squares`, enumeramos la secuencia de hojas del árbol, la filtramos para quedarnos solo con los números impares de la secuencia, elevamos al cuadrado cada elemento y sumamos los resultados:

(define (sum-odd-squares tree) (accumulate + 0 (map square (filter odd? (enumerate-tree tree)))))

Para `even/fibs`, enumeramos los enteros de 0 a *n*, generamos el número de Fibonacci para cada uno de estos enteros, filtramos la secuencia resultante para quedarnos solo con los elementos pares, y acumulamos los resultados en una lista:

(define (even-fibs n) (accumulate cons nil (filter even? (map fib (enumerate-interval 0 n)))))

El valor de expresar programas como operaciones de secuencia es que esto nos ayuda a hacer diseños de programas que son modulares, es decir, diseños que se construyen combinando piezas relativamente independientes. Podemos fomentar el diseño modular proporcionando una biblioteca de componentes estándar junto con una interfaz convencional para conectar los componentes de formas flexibles.

La construcción modular es una estrategia poderosa para controlar la complejidad en el diseño de ingeniería. En aplicaciones reales de procesamiento de señales, por ejemplo, los diseñadores construyen regularmente sistemas mediante la conexión en cascada de elementos seleccionados de familias estandarizadas de filtros y transductores. De manera similar, las operaciones de secuencia proporcionan una biblioteca de elementos de programa estándar que podemos mezclar y combinar. Por ejemplo, podemos reutilizar piezas de los procedimientos `sum/odd/squares` y `even/fibs` en un programa que construye una lista de los cuadrados de los primeros $n + 1$ números de Fibonacci:

(define (list-fib-squares n) (accumulate cons nil (map square (map fib (enumerate-interval 0 n))))) (list-fib-squares 10) *(0 1 1 4 9 25 64 169 441 1156 3025)*

Podemos reorganizar las piezas y usarlas para calcular el producto de los cuadrados de los enteros impares en una secuencia:

(define (product-of-squares-of-odd-elements sequence) (accumulate * 1 (map square (filter odd? sequence)))) (product-of-squares-of-odd-elements (list 1 2 3 4 5)) *225*

También podemos formular aplicaciones convencionales de procesamiento de datos en términos de operaciones de secuencia. Supongamos que tenemos una secuencia de registros de personal y queremos encontrar el salario del programador mejor pagado. Supongamos que tenemos un selector `salary` que devuelve el salario de un registro, y un predicado `programmer?` que

prueba si un registro es de un programador. Entonces podemos escribir

(define (salary-of-highest-paid-programmer records) (accumulate max 0 (map salary (filter programmer? records))))

Estos ejemplos dan solo una idea de la vasta gama de operaciones que se pueden expresar como operaciones de secuencia.[15]

Las secuencias, implementadas aquí como listas, sirven como una interfaz convencional que nos permite combinar módulos de procesamiento. Además, cuando representamos uniformemente las estructuras como secuencias, hemos localizado las dependencias de la estructura de datos en nuestros programas en un pequeño número de operaciones de secuencia. Al cambiar estas, podemos experimentar con representaciones alternativas de secuencias, dejando intacto el diseño general de nuestros programas. Explotaremos esta capacidad en Section 3.5, cuando generalicemos el paradigma de procesamiento de secuencias para admitir secuencias infinitas.

Ejercicio 2.33: Rellene las expresiones que faltan para completar las siguientes definiciones de algunas operaciones básicas de manipulación de listas como acumulaciones:

(define (map p sequence) (accumulate (lambda (x y) ⟨ ?? ⟩) nil sequence)) (define (append seq1 seq2) (accumulate cons ⟨ ?? ⟩ ⟨ ?? ⟩)) (define (length sequence) (accumulate ⟨ ?? ⟩ 0 sequence))

Ejercicio 2.34: La evaluación de un polinomio en x en un valor dado de x puede formularse como una acumulación. Evaluamos el polinomio

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

usando un algoritmo bien conocido llamado *regla de Horner*, que estructura el cálculo como

$$(\ldots(a_n x + a_{n-1})x + \ldots + a_1)x + a_0.$$

En otras palabras, comenzamos con a_n, multiplicamos por x, sumamos a_{n-1}, multiplicamos por x, y así sucesivamente, hasta que llegamos a a_0.[16]

Rellene la siguiente plantilla para producir un procedimiento que evalúe un polinomio usando la regla de Horner. Suponga que los coeficientes del polinomio están dispuestos en una secuencia, desde a_0 hasta a_n.

(define (horner-eval x coefficient-sequence) (accumulate (lambda (this-coeff higher-terms) ⟨ ?? ⟩) 0 coefficient-sequence))

Por ejemplo, para calcular $1 + 3x + 5x^3 + x^5$ en $x = 2$ usted evaluaría

(horner-eval 2 (list 1 3 0 5 0 1))

Ejercicio 2.35: Redefina `count/leaves` de Section 2.2.2 como una acumulación:

(define (count-leaves t) (accumulate ⟨ ?? ⟩ ⟨ ?? ⟩ (map ⟨ ?? ⟩ ⟨ ?? ⟩)))

Ejercicio 2.36: El procedimiento `accumulate/n` es similar a `accumulate` excepto que toma como tercer argumento una secuencia de secuencias, que se asume que todas tienen el mismo número de elementos. Aplica el procedimiento de acumulación designado para combinar todos los primeros elementos de las secuencias, todos los segundos elementos de las secuencias, y así sucesivamente, y devuelve una secuencia de los resultados. Por ejemplo, si `s` es una secuencia que contiene cuatro secuencias, `((1 2 3) (4 5 6) (7 8 9) (10 11 12))`, entonces el valor de `(accumulate/n + 0 s)` debería ser la secuencia `(22 26 30)`. Rellene las expresiones que faltan en la siguiente definición de `accumulate/n`:

(define (accumulate-n op init seqs) (if (null? (car seqs)) nil (cons (accumulate op init ⟨ ?? ⟩) (accumulate-n op init ⟨ ?? ⟩))))

Ejercicio 2.37: Supongamos que representamos los vectores $\hbox{\bf v} = (v_i)$ como secuencias de números, y las matrices $\hbox{\bf m} = (m_{i\!j})$ como secuencias de vectores (las filas de la matriz). Por ejemplo, la matriz

$$\left(\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 6 \\ 6 & 7 & 8 & 9 \end{array} \right)$$

se representa como la secuencia `((1 2 3 4) (4 5 6 6) (6 7 8 9))`. Con esta representación, podemos usar operaciones de secuencia para expresar de forma concisa las operaciones básicas de matrices y vectores. Estas operaciones (que se describen en cualquier libro de álgebra matricial) son las siguientes:

$$\begin{array}{rl} \hbox{\tt (dot-product v w)} & {\rm devuelve\;la\;suma\;} \Sigma_i v_i w_i; \\ \hbox{\tt (matrix-*-vector m v)} & {\rm devuelve\;el\;vector\;} \hbox{\bf t}, \\ & {\rm donde\;} t_i = \Sigma_{\kern-0.1em j} m_{i\!j} v_{\kern-0.1em j}; \\ \hbox{\tt (matrix-*-matrix m n)} & {\rm devuelve\;la\;matriz\;} \hbox{\bf p}, \\ & {\rm donde\;} p_{i\!j} = \Sigma_k m_{ik} n_{k\!j}; \\ \hbox{\tt (transpose m)} & {\rm devuelve\;la\;matriz\;} \hbox{\bf n}, \\ & {\rm donde\;}

n_{i\!j} = m_{\kern-0.1em ji}. \end{array}$$

Podemos definir el producto punto como[17]

(define (dot-product v w) (accumulate + 0 (map * v w)))

Rellene las expresiones que faltan en los siguientes procedimientos para calcular las otras operaciones de matriz. (El procedimiento `accumulate/n` se define en el Ejercicio 2.36.)

(define (matrix-*-vector m v) (map < ?? > m)) (define (transpose mat) (accumulate-n < ?? > < ?? > mat)) (define (matrix-*-matrix m n) (let ((cols (transpose n))) (map < ?? > m)))

Ejercicio 2.38: El procedimiento `accumulate` también se conoce como `fold/right`, porque combina el primer elemento de la secuencia con el resultado de combinar todos los elementos a la derecha. También hay un `fold/left`, que es similar a `fold/right`, excepto que combina los elementos trabajando en la dirección opuesta:

(define (fold-left op initial sequence) (define (iter result rest) (if (null? rest) result (iter (op result (car rest)) (cdr rest)))) (iter initial sequence))

¿Cuáles son los valores de

(fold-right / 1 (list 1 2 3)) (fold-left / 1 (list 1 2 3)) (fold-right list nil (list 1 2 3)) (fold-left list nil (list 1 2 3))

Dé una propiedad que `op` debería satisfacer para garantizar que `fold/right` y `fold/left` produzcan los mismos valores para cualquier secuencia.

Ejercicio 2.39: Complete las siguientes definiciones de `reverse` (Ejercicio 2.18) en términos de `fold/right` y `fold/left` del Ejercicio 2.38:

(define (reverse sequence) (fold-right (lambda (x y) < ?? >) nil sequence)) (define (reverse sequence) (fold-left (lambda (x y) < ?? >) nil sequence))

Mapeos Anidados

Podemos extender el paradigma de secuencia para incluir muchos cálculos que se expresan comúnmente usando bucles anidados.[18] Considere este problema: dado un entero positivo *n*, encuentre todos los pares ordenados de

enteros positivos distintos i y j, donde $1 \leq j < i \leq n$, tales que $i + j$ sea primo. Por ejemplo, si n es 6, entonces los pares son los siguientes:

$$\vbox{ \offinterlineskip \halign{ \strut \hfil \quad #\quad \hfil & \vrule \hfil \quad #\quad \hfil & \hfil \quad #\quad \hfil & \hfil \quad #\quad \hfil & \hfil \quad #\quad \hfil & \hfil \quad #\quad \hfil & \hfil \quad #\quad \hfil & \hfil \quad #\quad \hfil \cr i & 2 & 3 & 4 & 4 & 5 & 6 & 6 \cr j & 1 & 2 & 1 & 3 & 2 & 1 & 5 \cr \noalign{\hrule} $i + j$ & 3 & 5 & 5 & 7 & 7 & 7 & 11 \cr} }$$

Una forma natural de organizar este cálculo es generar la secuencia de todos los pares ordenados de enteros positivos menores o iguales a n, filtrar para seleccionar aquellos pares cuya suma sea prima, y luego, para cada par (i, j) que pase el filtro, producir la terna $(i, j, i + j)$.

Aquí hay una forma de generar la secuencia de pares: para cada entero $i \leq n$, enumere los enteros $j < i$, y para cada i y j, genere el par (i, j). En términos de operaciones de secuencia, mapeamos a lo largo de la secuencia `(enumerate/interval 1 n)`. Para cada i en esta secuencia, mapeamos a lo largo de la secuencia `(enumerate/interval 1 (- i 1))`. Para cada j en esta última secuencia, generamos el par `(list i j)`. Esto nos da una secuencia de pares para cada i. La combinación de todas las secuencias para todos los i (acumulando con `append`) produce la secuencia requerida de pares:[19]

(accumulate append nil (map (lambda (i) (map (lambda (j) (list i j)) (enumerate-interval 1 (- i 1)))) (enumerate-interval 1 n)))

La combinación de mapeo y acumulación con `append` es tan común en este tipo de programa que la aislaremos como un procedimiento separado:

(define (flatmap proc seq) (accumulate append nil (map proc seq)))

Ahora filtre esta secuencia de pares para encontrar aquellos cuya suma sea prima. El predicado de filtro se llama para cada elemento de la secuencia; su argumento es un par y debe extraer los enteros del par. Por lo tanto, el predicado a aplicar a cada elemento de la secuencia es

(define (prime-sum? pair) (prime? (+ (car pair) (cadr pair))))

Finalmente, genere la secuencia de resultados mapeando sobre los pares filtrados usando el siguiente procedimiento, que construye una terna que consiste en los dos elementos del par junto con su suma:

(define (make-pair-sum pair) (list (car pair) (cadr pair) (+ (car pair) (cadr pair))))

La combinación de todos estos pasos produce el procedimiento completo:

(define (prime-sum-pairs n) (map make-pair-sum (filter prime-sum? (flatmap (lambda (i) (map (lambda (j) (list i j)) (enumerate-interval 1 (- i 1)))) (enumerate-interval 1 n)))))

Los mapeos anidados también son útiles para secuencias distintas de las que enumeran intervalos. Supongamos que deseamos generar todas las permutaciones de un conjunto S; es decir, todas las formas de ordenar los elementos del conjunto. Por ejemplo, las permutaciones de $\{1, 2, 3\}$ son $\{1, 2, 3\}$, $\{1, 3, 2\}$, $\{2, 1, 3\}$, $\{2, 3, 1\}$, $\{3, 1, 2\}$ y $\{3, 2, 1\}$. Aquí hay un plan para generar las permutaciones de S: para cada elemento x en S, genere recursivamente la secuencia de permutaciones de $S - x$,[20] y añada x al principio de cada una. Esto produce, para cada x en S, la secuencia de permutaciones de S que comienzan con x. La combinación de estas secuencias para todos los x da todas las permutaciones de S:[21]

(define (permutations s) (if (null? s) ; conjunto vacío? (list nil) ; secuencia que contiene el conjunto vacío (flatmap (lambda (x) (map (lambda (p) (cons x p)) (permutations (remove x s)))) s)))

Observe cómo esta estrategia reduce el problema de generar permutaciones de S al problema de generar las permutaciones de conjuntos con menos elementos que S. En el caso terminal, descendemos hasta la lista vacía, que representa un conjunto sin elementos. Para esto, generamos `(list nil)`, que es una secuencia con un elemento, a saber, el conjunto sin elementos. El procedimiento `remove` utilizado en `permutations` devuelve todos los elementos en una secuencia dada excepto un elemento dado. Esto se puede expresar como un filtro simple:

(define (remove item sequence) (filter (lambda (x) (not (= x item))) sequence))

> **Ejercicio 2.40:** Defina un procedimiento `unique/pairs` que, dado un entero n, genere la secuencia de pares (i, j) con $1 \leq j < i \leq n$. Use `unique/pairs` para simplificar la definición de `prime/sum/pairs` dada anteriormente.
>
> **Ejercicio 2.41:** Escriba un procedimiento para encontrar todas las ternas ordenadas de enteros positivos distintos i, j y k menores o iguales a un entero dado n que sumen un entero dado s.
>
> **Ejercicio 2.42:** El "problema de las ocho reinas" pregunta cómo colocar ocho reinas en un tablero de ajedrez de modo que ninguna reina esté en jaque por ninguna otra (es decir, que no haya dos reinas en la misma fila,

columna o diagonal). Una posible solución se muestra en la Figura 2.8. Una forma de resolver el problema es trabajar a lo largo del tablero, colocando una reina en cada columna. Una vez que hemos colocado $k - 1$ reinas, debemos colocar la k^{th} reina en una posición donde no amenace a ninguna de las reinas que ya están en el tablero. Podemos formular este enfoque de forma recursiva: supongamos que ya hemos generado la secuencia de todas las formas posibles de colocar $k - 1$ reinas en las primeras $k - 1$ columnas del tablero. Para cada una de estas formas, genere un conjunto extendido de posiciones colocando una reina en cada fila de la k^{th} columna. Ahora filtre estas, manteniendo solo las posiciones para las cuales la reina en la k^{th} columna está a salvo con respecto a las otras reinas. Esto produce la secuencia de todas las formas de colocar k reinas en las primeras k columnas. Al continuar este proceso, produciremos no solo una solución, sino todas las soluciones al problema.

Figura 2.8: Una solución al problema de las ocho reinas.

Implementamos esta solución como un procedimiento `queens`, que devuelve una secuencia de todas las soluciones al problema de colocar n reinas en un tablero de ajedrez de $n \times n$. `queens` tiene un procedimiento interno `queen/cols` que devuelve la secuencia de todas las formas de colocar reinas en las primeras k columnas del tablero.

(define (queens board-size) (define (queen-cols k) (if (= k 0) (list empty-board) (filter (lambda (positions) (safe? k positions)) (flatmap (lambda (rest-of-queens) (map (lambda (new-row) (adjoin-position new-row k rest-of-queens)) (enumerate-interval 1 board-size))) (queen-cols (- k 1)))))) (queen-cols board-size))

En este procedimiento, `rest/of/queens` es una forma de colocar $k - 1$ reinas en las primeras $k - 1$ columnas, y `new/row` es una fila propuesta en la que colocar la reina para la k^{th} columna. Complete el programa implementando la representación para conjuntos de posiciones del tablero, incluido el procedimiento `adjoin/position`, que añade una nueva posición de fila-columna a un conjunto de posiciones, y `empty/board`, que representa un conjunto vacío de posiciones. También debe escribir el procedimiento `safe?`, que determina para un conjunto de posiciones, si la reina en la k^{th} columna está a salvo con respecto a las demás. (Tenga en cuenta que solo necesitamos comprobar si la nueva reina está a salvo; ya se garantiza que las otras reinas están a salvo entre

sí).

Ejercicio 2.43: Louis Reasoner está teniendo un terrible problema con el Ejercicio 2.42. Su procedimiento `queens` parece funcionar, pero se ejecuta extremadamente lento. (Louis nunca logra esperar lo suficiente para que resuelva siquiera el caso de 6×6). Cuando Louis le pide ayuda a Eva Lu Ator, ella le señala que ha intercambiado el orden de los mapeos anidados en el `flatmap`, escribiéndolo como

(flatmap (lambda (new-row) (map (lambda (rest-of-queens) (adjoin-position new-row k rest-of-queens)) (queen-cols (- k 1)))) (enumerate-interval 1 board-size))

Explique por qué este intercambio hace que el programa se ejecute lentamente. Estime cuánto tiempo le tomará al programa de Louis resolver el problema de las ocho reinas, suponiendo que el programa del Ejercicio 2.42 resuelve el problema en un tiempo *T*.

Ejemplo: Un Lenguaje de Imágenes

Esta sección presenta un lenguaje simple para dibujar imágenes que ilustra el poder de la abstracción de datos y la clausura, y también explota procedimientos de orden superior de una manera esencial. El lenguaje está diseñado para facilitar la experimentación con patrones como los de la Figura 2.9, que están compuestos de elementos repetidos que se desplazan y se escalan.[22] En este lenguaje, los objetos de datos que se combinan se representan como procedimientos en lugar de como estructuras de lista. Así como `cons`, que satisface la propiedad de clausura, nos permitió construir fácilmente estructuras de lista arbitrariamente complicadas, las operaciones en este lenguaje, que también satisfacen la propiedad de clausura, nos permiten construir fácilmente patrones arbitrariamente complicados.

El lenguaje de imágenes

Cuando comenzamos nuestro estudio de la programación en Section 1.1, enfatizamos la importancia de describir un lenguaje centrándonos en las primitivas del lenguaje, sus medios de combinación y sus medios de abstracción. Seguiremos ese marco aquí.

Figura 2.9: Diseños generados con el lenguaje de imágenes.

Figura 2.10: Imágenes producidas por el pintor `wave`, con respecto a cuatro marcos diferentes. Los marcos, mostrados con líneas punteadas, no forman parte de las imágenes.

Parte de la elegancia de este lenguaje de imágenes es que solo hay un tipo de elemento, llamado *pintor*. Un pintor dibuja una imagen que se desplaza y se escala para que quepa dentro de un marco designado en forma de paralelogramo. Por ejemplo, hay un pintor primitivo que llamaremos `wave` que hace un dibujo de líneas crudo, como se muestra en la Figura 2.10. La forma real del dibujo depende del marco: las cuatro imágenes de la figura 2.10 son producidas por el mismo pintor `wave`, pero con respecto a cuatro marcos diferentes. Los pintores pueden ser más elaborados que esto: el pintor primitivo llamado `rogers` pinta una imagen del fundador del MIT, William Barton Rogers, como se muestra en la Figura 2.11.[23] Las cuatro imágenes de la figura 2.11 se dibujan con respecto a los mismos cuatro marcos que las imágenes de `wave` de la figura 2.10.

Figura 2.11: Imágenes de William Barton Rogers, fundador y primer presidente del MIT, pintadas con respecto a los mismos cuatro marcos que en la Figura 2.10 (imagen original de Wikimedia Commons).

Para combinar imágenes, usamos varias operaciones que construyen nuevos pintores a partir de pintores dados. Por ejemplo, la operación `beside` toma dos pintores y produce un pintor nuevo y compuesto que dibuja la imagen del primer pintor en la mitad izquierda del marco y la imagen del segundo pintor en la mitad derecha del marco. De manera similar, `below` toma dos pintores y produce un pintor compuesto que dibuja la imagen del primer pintor debajo de la imagen del segundo pintor. Algunas operaciones transforman un solo pintor para producir un nuevo pintor. Por ejemplo, `flip/vert` toma un pintor y produce un pintor que dibuja su imagen al revés, y `flip/horiz` produce un pintor que dibuja la imagen del pintor original invertida de izquierda a derecha.

La Figura 2.12 muestra el dibujo de un pintor llamado `wave4` que se construye en dos etapas a partir de `wave`:

(define wave2 (beside wave (flip-vert wave))) (define wave4 (below wave2 wave2))

Figura 2.12: Creación de una figura compleja, a partir del pintor `wave` de la Figura 2.10.

Al construir una imagen compleja de esta manera, estamos explotando el hecho de que los pintores están cerrados bajo los medios de combinación del lenguaje. El `beside` o `below` de dos pintores es a su vez un pintor; por lo tanto, podemos usarlo como un elemento para hacer pintores más complejos. Al igual que con la construcción de estructuras de lista usando `cons`, la clausura de nuestros datos bajo los medios de combinación es crucial para la capacidad de crear estructuras complejas usando solo unas pocas operaciones.

Una vez que podemos combinar pintores, nos gustaría poder abstraer patrones típicos de combinación de pintores. Implementaremos las operaciones de pintor como procedimientos de Scheme. Esto significa que no necesitamos un mecanismo de abstracción especial en el lenguaje de imágenes: dado que los medios de combinación son procedimientos ordinarios de Scheme, automáticamente tenemos la capacidad de hacer cualquier cosa con las operaciones de pintor que podemos hacer con los procedimientos. Por ejemplo, podemos abstraer el patrón en `wave4` como

(define (flipped-pairs painter) (let ((painter2 (beside painter (flip-vert painter)))) (below painter2 painter2)))

y definir `wave4` como una instancia de este patrón:

(define wave4 (flipped-pairs wave))

Figura 2.13: Planes recursivos para `right/split` y `corner/split`.

También podemos definir operaciones recursivas. Aquí hay una que hace que los pintores se dividan y se ramifiquen hacia la derecha como se muestra en la Figura 2.13 y la Figura 2.14:

(define (right-split painter n) (if (= n 0) painter (let ((smaller (right-split painter (- n 1)))) (beside painter (below smaller smaller)))))

Podemos producir patrones equilibrados ramificando hacia arriba y hacia la derecha (ver ejercicio Ejercicio 2.44 y las figuras Figura 2.13 y Figura 2.14):

(define (corner-split painter n) (if (= n 0) painter (let ((up (up-split painter (- n 1))) (right (right-split painter (- n 1)))) (let ((top-left (beside up up)) (bottom-right (below right right)) (corner (corner-split painter (- n 1)))) (beside (below painter top-left) (below bottom-right corner))))))

Colocando cuatro copias de un `corner/split` apropiadamente, obtenemos un

patrón llamado `square/limit`, cuya aplicación a `wave` y `rogers` se muestra en la Figura 2.9:

```
(define (square-limit painter n) (let ((quarter (corner-split painter n))) (let ((half (beside (flip-horiz quarter) quarter))) (below (flip-vert half) half))))
```

Ejercicio 2.44: Defina el procedimiento `up/split` utilizado por `corner/split`. Es similar a `right/split`, excepto que intercambia los roles de `below` y `beside`.

Figura 2.14: Las operaciones recursivas `right/split` y `corner/split` aplicadas a los pintores `wave` y `rogers`. La combinación de cuatro figuras de `corner/split` produce diseños simétricos de `square/limit` como se muestra en la Figura 2.9.

Operaciones de orden superior

Además de abstraer patrones de combinación de pintores, podemos trabajar a un nivel superior, abstrayendo patrones de combinación de operaciones de pintor. Es decir, podemos ver las operaciones de pintor como elementos a manipular y podemos escribir medios de combinación para estos elementos, procedimientos que toman operaciones de pintor como argumentos y crean nuevas operaciones de pintor.

Por ejemplo, `flipped/pairs` y `square/limit` organizan cada uno cuatro copias de la imagen de un pintor en un patrón cuadrado; solo difieren en cómo orientan las copias. Una forma de abstraer este patrón de combinación de pintores es con el siguiente procedimiento, que toma cuatro operaciones de pintor de un argumento y produce una operación de pintor que transforma un pintor dado con esas cuatro operaciones y organiza los resultados en un cuadrado. `tl`, `tr`, `bl` y `br` son las transformaciones que se aplicarán a la copia superior izquierda, la copia superior derecha, la copia inferior izquierda y la copia inferior derecha, respectivamente.

```
(define (square-of-four tl tr bl br) (lambda (painter) (let ((top (beside (tl painter) (tr painter))) (bottom (beside (bl painter) (br painter)))) (below bottom top))))
```

Entonces `flipped/pairs` puede definirse en términos de `square/of/four` de la siguiente manera:[24]

```
(define (flipped-pairs painter) (let ((combine4 (square-of-four identity flip-vert identity flip-vert))) (combine4 painter)))
```

y `square/limit` puede expresarse como[25]

(define (square-limit painter n) (let ((combine4 (square-of-four flip-horiz identity rotate180 flip-vert))) (combine4 (corner-split painter n))))

Ejercicio 2.45: `right/split` y `up/split` pueden expresarse como instancias de una operación general de división. Defina un procedimiento `split` con la propiedad de que al evaluar

(define right-split (split beside below)) (define up-split (split below beside))

produce procedimientos `right/split` y `up/split` con los mismos comportamientos que los ya definidos.

Marcos

Antes de que podamos mostrar cómo implementar pintores y sus medios de combinación, primero debemos considerar los marcos. Un marco puede describirse mediante tres vectores: un vector de origen y dos vectores de borde. El vector de origen especifica el desplazamiento del origen del marco desde algún origen absoluto en el plano, y los vectores de borde especifican los desplazamientos de las esquinas del marco desde su origen. Si los bordes son perpendiculares, el marco será rectangular. De lo contrario, el marco será un paralelogramo más general.

La Figura 2.15 muestra un marco y sus vectores asociados. De acuerdo con la abstracción de datos, no necesitamos ser específicos todavía sobre cómo se representan los marcos, aparte de decir que hay un constructor `make/frame`, que toma tres vectores y produce un marco, y tres selectores correspondientes `origin/frame`, `edge1/frame` y `edge2/frame` (ver Ejercicio 2.47).

Figura 2.15: Un marco se describe mediante tres vectores: un origen y dos bordes.

Usaremos coordenadas en el cuadrado unitario ($0 \leq x, y \leq 1$) para especificar imágenes. Con cada marco, asociamos un *mapa de coordenadas del marco*, que se usará para desplazar y escalar imágenes para que se ajusten al marco. El mapa transforma el cuadrado unitario en el marco mapeando el vector $\hbox{\bf v} = (x, y)$ a la suma vectorial

$${\rm Origen(Marco)} + x \cdot {\rm Borde_1(Marco)} + y \cdot {\rm

Borde_2(Marco)}.$$

Por ejemplo, (0, 0) se mapea al origen del marco, (1, 1) al vértice diagonalmente opuesto al origen, y (0.5, 0.5) al centro del marco. Podemos crear el mapa de coordenadas de un marco con el siguiente procedimiento:[26]

(define (frame-coord-map frame) (lambda (v) (add-vect (origin-frame frame) (add-vect (scale-vect (xcor-vect v) (edge1-frame frame)) (scale-vect (ycor-vect v) (edge2-frame frame))))))

Observe que aplicar `frame/coord/map` a un marco devuelve un procedimiento que, dado un vector, devuelve un vector. Si el vector de argumento está en el cuadrado unitario, el vector resultante estará en el marco. Por ejemplo,

((frame-coord-map a-frame) (make-vect 0 0))

devuelve el mismo vector que

(origin-frame a-frame)

Ejercicio 2.46: Un vector bidimensional $\hbox{\bf v}$ que va desde el origen hasta un punto puede representarse como un par que consiste en una coordenada *x* y una coordenada *y*. Implemente una abstracción de datos para vectores dando un constructor `make/vect` y los selectores correspondientes `xcor/vect` e `ycor/vect`. En términos de sus selectores y constructor, implemente los procedimientos `add/vect`, `sub/vect` y `scale/vect` que realizan las operaciones de suma de vectores, resta de vectores y multiplicación de un vector por un escalar:

$$\begin{array}{r@{{}={}}l} (x_1, y_1) + (x_2, y_2) & (x_1 + x_2, y_1 + y_2), \\ (x_1, y_1) - (x_2, y_2) & (x_1 - x_2, y_1 - y_2), \\ s \cdot (x, y) & (sx, sy). \end{array}$$

Ejercicio 2.47: Aquí hay dos posibles constructores para marcos:

(define (make-frame origin edge1 edge2) (list origin edge1 edge2))
(define (make-frame origin edge1 edge2) (cons origin (cons edge1 edge2)))

Para cada constructor, proporcione los selectores apropiados para producir una implementación para marcos.

Pintores

Un pintor se representa como un procedimiento que, dado un marco como argumento, dibuja una imagen particular desplazada y escalada para que se ajuste al marco. Es decir, si `p` es un pintor y `f` es un marco, entonces producimos la imagen de `p` en `f` llamando a `p` con `f` como argumento.

Los detalles de cómo se implementan los pintores primitivos dependen de las características particulares del sistema de gráficos y del tipo de imagen que se va a dibujar. Por ejemplo, supongamos que tenemos un procedimiento `draw/line` que dibuja una línea en la pantalla entre dos puntos especificados. Entonces podemos crear pintores para dibujos de líneas, como el pintor `wave` en la Figura 2.10, a partir de listas de segmentos de línea de la siguiente manera:[27]

(define (segments->painter segment-list) (lambda (frame) (for-each (lambda (segment) (draw-line ((frame-coord-map frame) (start-segment segment)) ((frame-coord-map frame) (end-segment segment)))) segment-list)))

Los segmentos se dan usando coordenadas con respecto al cuadrado unitario. Para cada segmento de la lista, el pintor transforma los puntos finales del segmento con el mapa de coordenadas del marco y dibuja una línea entre los puntos transformados.

Representar a los pintores como procedimientos erige una poderosa barrera de abstracción en el lenguaje de imágenes. Podemos crear e mezclar todo tipo de pintores primitivos, basados en una variedad de capacidades gráficas. Los detalles de su implementación no importan. Cualquier procedimiento puede servir como pintor, siempre que tome un marco como argumento y dibuje algo escalado para que se ajuste al marco.[28]

Ejercicio 2.48: Un segmento de línea dirigido en el plano puede representarse como un par de vectores: el vector que va desde el origen hasta el punto de inicio del segmento, y el vector que va desde el origen hasta el punto final del segmento. Use su representación de vectores del Ejercicio 2.46 para definir una representación para segmentos con un constructor `make/segment` y los selectores `start/segment` y `end/segment`.

Ejercicio 2.49: Use `segments->painter` para definir los siguientes pintores primitivos:

1. El pintor que dibuja el contorno del marco designado.

2. El pintor que dibuja una "X" conectando las esquinas opuestas del marco.

3. El pintor que dibuja una forma de diamante conectando los puntos medios de los lados del marco.

4. El pintor `wave`.

Transformando y combinando pintores

Una operación sobre pintores (como `flip/vert` o `beside`) funciona creando un pintor que invoca a los pintores originales con respecto a los marcos derivados del marco del argumento. Así, por ejemplo, `flip/vert` no necesita saber cómo funciona un pintor para voltearlo, solo necesita saber cómo voltear un marco: el pintor volteado simplemente usa el pintor original, pero en el marco invertido.

Las operaciones de pintor se basan en el procedimiento `transform/painter`, que toma como argumentos un pintor e información sobre cómo transformar un marco y produce un nuevo pintor. El pintor transformado, cuando se le llama en un marco, transforma el marco y llama al pintor original en el marco transformado. Los argumentos de `transform/painter` son puntos (representados como vectores) que especifican las esquinas del nuevo marco: cuando se mapean en el marco, el primer punto especifica el origen del nuevo marco y los otros dos especifican los extremos de sus vectores de borde. Por lo tanto, los argumentos dentro del cuadrado unitario especifican un marco contenido dentro del marco original.

(define (transform-painter painter origin corner1 corner2) (lambda (frame) (let ((m (frame-coord-map frame))) (let ((new-origin (m origin))) (painter (make-frame new-origin (sub-vect (m corner1) new-origin) (sub-vect (m corner2) new-origin)))))))

Así es como se voltean verticalmente las imágenes de los pintores:

(define (flip-vert painter) (transform-painter painter (make-vect 0.0 1.0) ; nuevo `origen` (make-vect 1.0 1.0) ; nuevo final de `borde1` (make-vect 0.0 0.0))) ; nuevo final de `borde2`

Usando `transform/painter`, podemos definir fácilmente nuevas transformaciones. Por ejemplo, podemos definir un pintor que reduce su imagen al cuarto superior derecho del marco que se le da:

(define (shrink-to-upper-right painter) (transform-painter painter (make-vect 0.5 0.5) (make-vect 1.0 0.5) (make-vect 0.5 1.0)))

Otras transformaciones rotan las imágenes 90 grados en sentido antihorario[29]

```
(define (rotate90 painter) (transform-painter painter (make-vect 1.0 0.0) (make-vect 1.0 1.0) (make-vect 0.0 0.0)))
```

o aplastan las imágenes hacia el centro del marco:[30]

```
(define (squash-inwards painter) (transform-painter painter (make-vect 0.0 0.0) (make-vect 0.65 0.35) (make-vect 0.35 0.65)))
```

La transformación del marco también es la clave para definir los medios para combinar dos o más pintores. El procedimiento `beside`, por ejemplo, toma dos pintores, los transforma para pintar en las mitades izquierda y derecha de un marco de argumento respectivamente, y produce un nuevo pintor compuesto. Cuando se le da un marco al pintor compuesto, llama al primer pintor transformado para pintar en la mitad izquierda del marco y llama al segundo pintor transformado para pintar en la mitad derecha del marco:

```
(define (beside painter1 painter2) (let ((split-point (make-vect 0.5 0.0))) (let ((paint-left (transform-painter painter1 (make-vect 0.0 0.0) split-point (make-vect 0.0 1.0))) (paint-right (transform-painter painter2 split-point (make-vect 1.0 0.0) (make-vect 0.5 1.0)))) (lambda (frame) (paint-left frame) (paint-right frame)))))
```

Observe cómo la abstracción de datos del pintor, y en particular la representación de los pintores como procedimientos, facilita la implementación de `beside`. El procedimiento `beside` no necesita saber nada sobre los detalles de los pintores componentes, aparte de que cada pintor dibujará algo en su marco designado.

> **Ejercicio 2.50:** Defina la transformación `flip/horiz`, que voltea los pintores horizontalmente, y las transformaciones que rotan los pintores 180 grados y 270 grados en sentido antihorario.
>
> **Ejercicio 2.51:** Defina la operación `below` para pintores. `below` toma dos pintores como argumentos. El pintor resultante, dado un marco, dibuja con el primer pintor en la parte inferior del marco y con el segundo pintor en la parte superior. Defina `below` de dos maneras diferentes: primero escribiendo un procedimiento que sea análogo al procedimiento `beside` dado anteriormente, y de nuevo en términos de `beside` y operaciones de rotación adecuadas (del Ejercicio 2.50).

Niveles de lenguaje para un diseño robusto

El lenguaje de imágenes ejercita algunas de las ideas críticas que hemos introducido sobre la abstracción con procedimientos y datos. Las

abstracciones de datos fundamentales, los pintores, se implementan utilizando representaciones procedimentales, lo que permite al lenguaje manejar diferentes capacidades de dibujo básicas de manera uniforme. Los medios de combinación satisfacen la propiedad de clausura, lo que nos permite construir fácilmente diseños complejos. Finalmente, todas las herramientas para abstraer procedimientos están a nuestra disposición para abstraer los medios de combinación de los pintores.

También hemos vislumbrado otra idea crucial sobre los lenguajes y el diseño de programas. Este es el enfoque del *diseño estratificado*, la noción de que un sistema complejo debe estructurarse como una secuencia de niveles que se describen utilizando una secuencia de lenguajes. Cada nivel se construye combinando partes que se consideran primitivas en ese nivel, y las partes construidas en cada nivel se utilizan como primitivas en el siguiente nivel. El lenguaje utilizado en cada nivel de un diseño estratificado tiene primitivas, medios de combinación y medios de abstracción apropiados para ese nivel de detalle.

El diseño estratificado impregna la ingeniería de sistemas complejos. Por ejemplo, en la ingeniería informática, los resistores y transistores se combinan (y se describen usando un lenguaje de circuitos analógicos) para producir partes como compuertas Y y compuertas O, que forman las primitivas de un lenguaje para el diseño de circuitos digitales.[31] Estas partes se combinan para construir procesadores, estructuras de bus y sistemas de memoria, que a su vez se combinan para formar ordenadores, utilizando lenguajes apropiados para la arquitectura de ordenadores. Los ordenadores se combinan para formar sistemas distribuidos, utilizando lenguajes apropiados para describir las interconexiones de red, y así sucesivamente.

Como un pequeño ejemplo de estratificación, nuestro lenguaje de imágenes utiliza elementos primitivos (pintores primitivos) que se crean utilizando un lenguaje que especifica puntos y líneas para proporcionar las listas de segmentos de línea para `segments/>painter`, o los detalles de sombreado para un pintor como `rogers`. La mayor parte de nuestra descripción del lenguaje de imágenes se centró en la combinación de estas primitivas, utilizando combinadores geométricos como `beside` y `below`. También trabajamos a un nivel superior, considerando `beside` y `below` como primitivas que se manipularán en un lenguaje cuyas operaciones, como `square/of/four`, capturan patrones comunes de combinación de combinadores geométricos.

El diseño estratificado ayuda a que los programas sean *robustos*, es decir, hace que sea probable que pequeños cambios en una especificación requieran cambios correspondientemente pequeños en el programa. Por ejemplo,

supongamos que quisiéramos cambiar la imagen basada en `wave` que se muestra en la Figura 2.9. Podríamos trabajar en el nivel más bajo para cambiar la apariencia detallada del elemento `wave`; podríamos trabajar en el nivel medio para cambiar la forma en que `corner/split` replica el `wave`; podríamos trabajar en el nivel más alto para cambiar cómo `square/limit` organiza las cuatro copias de la esquina. En general, cada nivel de un diseño estratificado proporciona un vocabulario diferente para expresar las características del sistema y un tipo diferente de capacidad para cambiarlo.

Ejercicio 2.52: Realice cambios en el límite cuadrado de `wave` que se muestra en la Figura 2.9 trabajando en cada uno de los niveles descritos anteriormente. En particular:

1. Agregue algunos segmentos al pintor primitivo `wave` del Ejercicio 2.49 (para agregar una sonrisa, por ejemplo).

2. Cambie el patrón construido por `corner/split` (por ejemplo, usando solo una copia de las imágenes `up/split` y `right/split` en lugar de dos).

3. Modifique la versión de `square/limit` que usa `square/of/four` para ensamblar las esquinas en un patrón diferente. (Por ejemplo, podría hacer que el gran Sr. Rogers mire hacia afuera desde cada esquina del cuadrado).

Datos Simbólicos

Todos los objetos de datos compuestos que hemos utilizado hasta ahora se construyeron finalmente a partir de números. En esta sección extendemos la capacidad de representación de nuestro lenguaje introduciendo la capacidad de trabajar con símbolos arbitrarios como datos.

Cita

Si podemos formar datos compuestos usando símbolos, podemos tener listas como

(a b c d) (23 45 17) ((Norah 12) (Molly 9) (Anna 7) (Lauren 6) (Charlotte 4))

Las listas que contienen símbolos pueden parecerse a las expresiones de nuestro lenguaje:

(* (+ 23 45) (+ x 9)) (define (fact n) (if (= n 1) 1 (* n (fact (- n 1)))))

Para manipular símbolos, necesitamos un nuevo elemento en nuestro lenguaje: la capacidad de *citar* un objeto de datos. Supongamos que queremos construir la lista `(a b)`. No podemos lograr esto con `(list a b)`, porque esta expresión construye una lista de los *valores* de `a` y `b` en lugar de los símbolos mismos. Este problema es bien conocido en el contexto de los lenguajes naturales, donde las palabras y las oraciones pueden considerarse entidades semánticas o cadenas de caracteres (entidades sintácticas). La práctica común en los lenguajes naturales es usar comillas para indicar que una palabra o una oración debe tratarse literalmente como una cadena de caracteres. Por ejemplo, la primera letra de "John" es claramente "J". Si le decimos a alguien "di tu nombre en voz alta", esperamos escuchar el nombre de esa persona. Sin embargo, si le decimos a alguien "di 'tu nombre' en voz alta", esperamos escuchar las palabras "tu nombre". Tenga en cuenta que nos vemos obligados a anidar comillas para describir lo que otra persona podría decir.[32]

Podemos seguir esta misma práctica para identificar listas y símbolos que deben tratarse como objetos de datos en lugar de como expresiones a evaluar. Sin embargo, nuestro formato para citar difiere del de los lenguajes naturales en que colocamos una comilla (tradicionalmente, el símbolo de comilla simple `'`) solo al principio del objeto que se va a citar. Podemos salirnos con la nuestra en la sintaxis de Scheme porque confiamos en los espacios en blanco y los paréntesis para delimitar los objetos. Por lo tanto, el significado del carácter de comilla simple es citar el siguiente objeto.[33]

Ahora podemos distinguir entre los símbolos y sus valores:

(define a 1) (define b 2) (list a b) *(1 2)* (list 'a 'b) *(a b)* (list 'a b) *(a 2)*

La cita también nos permite escribir objetos compuestos, utilizando la representación impresa convencional para las listas:[34]

(car '(a b c)) *a* (cdr '(a b c)) *(b c)*

De acuerdo con esto, podemos obtener la lista vacía evaluando `'()`, y así prescindir de la variable `nil`.

Una primitiva adicional utilizada en la manipulación de símbolos es `eq?`, que toma dos símbolos como argumentos y prueba si son el mismo.[35] Usando `eq?`, podemos implementar un procedimiento útil llamado `memq`. Este toma dos argumentos, un símbolo y una lista. Si el símbolo no está contenido en la lista (es decir, no es `eq?` a ningún elemento de la lista), entonces `memq` devuelve falso. De lo contrario, devuelve la sublista de la lista que comienza con la primera aparición del símbolo:

(define (memq item x) (cond ((null? x) false) ((eq? item (car x)) x) (else (memq item (cdr x)))))

Por ejemplo, el valor de

(memq 'apple '(pear banana prune))

es falso, mientras que el valor de

(memq 'apple '(x (apple sauce) y apple pear))

es `(apple pear)`.

Ejercicio 2.53: ¿Qué imprimiría el intérprete en respuesta a la evaluación de cada una de las siguientes expresiones?

(list 'a 'b 'c) (list (list 'george)) (cdr '((x1 x2) (y1 y2))) (cadr '((x1 x2) (y1 y2))) (pair? (car '(a short list))) (memq 'red '((red shoes) (blue socks))) (memq 'red '(red shoes blue socks))

Ejercicio 2.54: Se dice que dos listas son `equal?` si contienen elementos iguales dispuestos en el mismo orden. Por ejemplo,

(equal? '(this is a list) '(this is a list))

es verdadero, pero

(equal? '(this is a list) '(this (is a) list))

es falso. Para ser más precisos, podemos definir `equal?` recursivamente en términos de la igualdad básica de símbolos `eq?` diciendo que `a` y `b` son `equal?` si ambos son símbolos y los símbolos son `eq?`, o si ambos son listas tales que `(car a)` es `equal?` a `(car b)` y `(cdr a)` es `equal?` a `(cdr b)`. Usando esta idea, implemente `equal?` como un procedimiento.[36]

Ejercicio 2.55: Eva Lu Ator escribe al intérprete la expresión

(car 'abracadabra)

Para su sorpresa, el intérprete imprime de vuelta `quote`. Explique.

Ejemplo: Diferenciación Simbólica

Como ilustración de la manipulación de símbolos y una ilustración adicional

de la abstracción de datos, considere el diseño de un procedimiento que realiza la diferenciación simbólica de expresiones algebraicas. Nos gustaría que el procedimiento tomara como argumentos una expresión algebraica y una variable y devolviera la derivada de la expresión con respecto a la variable. Por ejemplo, si los argumentos del procedimiento son $ax^2 + bx + c$ y x, el procedimiento debería devolver $2ax + b$. La diferenciación simbólica tiene una importancia histórica especial en Lisp. Fue uno de los ejemplos motivadores detrás del desarrollo de un lenguaje de ordenador para la manipulación de símbolos. Además, marcó el comienzo de la línea de investigación que condujo al desarrollo de potentes sistemas para el trabajo matemático simbólico, que actualmente están siendo utilizados por un número creciente de matemáticos aplicados y físicos.

Al desarrollar el programa de diferenciación simbólica, seguiremos la misma estrategia de abstracción de datos que seguimos al desarrollar el sistema de números racionales de Section 2.1.1. Es decir, primero definiremos un algoritmo de diferenciación que opera sobre objetos abstractos como "sumas", "productos" y "variables" sin preocuparnos de cómo se representarán. Solo después abordaremos el problema de la representación.

El programa de diferenciación con datos abstractos

Para mantener las cosas simples, consideraremos un programa de diferenciación simbólica muy simple que maneja expresiones que se construyen usando solo las operaciones de suma y multiplicación con dos argumentos. La diferenciación de cualquier expresión de este tipo se puede llevar a cabo aplicando las siguientes reglas de reducción:

$${{\it dc} \over {\it dx}} = 0, \quad {\rm para\ } c\ {\rm una\ constante\ o\ una\ variable\ diferente\ de\ } x,$$

$${{\it dx} \over {\it dx}} = 1,$$

$${{\it d\,(u + v\,)} \over {\it dx}} = {{\it du} \over {\it dx}} + {{\it dv} \over {\it dx}},$$

$${{\it d\,(uv\,)} \over {\it dx}} = u {{\it dv} \over {\it dx}} + v {{\it du} \over {\it dx}}.$$

Observe que las dos últimas reglas son de naturaleza recursiva. Es decir, para obtener la derivada de una suma, primero encontramos las derivadas de los términos y las sumamos. Cada uno de los términos puede a su vez ser una expresión que necesita ser descompuesta. La descomposición en piezas cada vez más pequeñas producirá finalmente piezas que son constantes o

variables, cuyas derivadas serán 0 o 1.

Para incorporar estas reglas en un procedimiento, nos permitimos un poco de pensamiento desiderativo, como hicimos al diseñar la implementación de los números racionales. Si tuviéramos un medio para representar expresiones algebraicas, deberíamos poder decir si una expresión es una suma, un producto, una constante o una variable. Deberíamos poder extraer las partes de una expresión. Para una suma, por ejemplo, queremos poder extraer el sumando (primer término) y el sumando (segundo término). También deberíamos poder construir expresiones a partir de partes. Supongamos que ya tenemos procedimientos para implementar los siguientes selectores, constructores y predicados:

(variable? e) ¿Es `e` una variable? (same-variable? v1 v2) ¿Son `v1` y `v2` la misma variable? (sum? e) ¿Es `e` una suma? (addend e) Sumando de la suma `e`. (augend e) Sumando de la suma `e`. (make-sum a1 a2) Construye la suma de `a1` y `a2`. (product? e) ¿Es `e` un producto? (multiplier e) Multiplicador del producto `e`. (multiplicand e) Multiplicando del producto `e`. (make-product m1 m2) Construye el producto de `m1` y `m2`.

Usando estos, y el predicado primitivo `number?`, que identifica números, podemos expresar las reglas de diferenciación como el siguiente procedimiento:

(define (deriv exp var) (cond ((number? exp) 0) ((variable? exp) (if (same-variable? exp var) 1 0)) ((sum? exp) (make-sum (deriv (addend exp) var) (deriv (augend exp) var))) ((product? exp) (make-sum (make-product (multiplier exp) (deriv (multiplicand exp) var)) (make-product (deriv (multiplier exp) var) (multiplicand exp)))) (else (error "unknown expression type: DERIV" exp))))

Este procedimiento `deriv` incorpora el algoritmo de diferenciación completo. Dado que se expresa en términos de datos abstractos, funcionará sin importar cómo elijamos representar las expresiones algebraicas, siempre que diseñemos un conjunto adecuado de selectores y constructores. Este es el problema que debemos abordar a continuación.

Representación de expresiones algebraicas

Podemos imaginar muchas formas de usar la estructura de lista para representar expresiones algebraicas. Por ejemplo, podríamos usar listas de símbolos que reflejen la notación algebraica habitual, representando $ax + b$ como la lista `(a * x + b)`. Sin embargo, una opción especialmente sencilla es usar la misma notación prefija entre paréntesis que Lisp usa para las

combinaciones; es decir, representar $ax + b$ como `(+ (* a x) b)`. Entonces, nuestra representación de datos para el problema de la diferenciación es la siguiente:

- Las variables son símbolos. Se identifican mediante el predicado primitivo `symbol?`:

  (define (variable? x) (symbol? x))

- Dos variables son iguales si los símbolos que las representan son `eq?`:

  (define (same-variable? v1 v2) (and (variable? v1) (variable? v2) (eq? v1 v2)))

- Las sumas y los productos se construyen como listas:

  (define (make-sum a1 a2) (list '+ a1 a2)) (define (make-product m1 m2) (list '* m1 m2))

- Una suma es una lista cuyo primer elemento es el símbolo `+`:

  (define (sum? x) (and (pair? x) (eq? (car x) '+)))

- El sumando es el segundo elemento de la lista de suma:

  (define (addend s) (cadr s))

- El sumando es el tercer elemento de la lista de suma:

  (define (augend s) (caddr s))

- Un producto es una lista cuyo primer elemento es el símbolo `*`:

  (define (product? x) (and (pair? x) (eq? (car x) '*)))

- El multiplicador es el segundo elemento de la lista de producto:

  (define (multiplier p) (cadr p))

- El multiplicando es el tercer elemento de la lista de producto:

  (define (multiplicand p) (caddr p))

Así, solo necesitamos combinar esto con el algoritmo tal como se plasma en `deriv` para tener un programa de diferenciación simbólica que funcione. Veamos algunos ejemplos de su comportamiento:

(deriv '(+ x 3) 'x) *(+ 1 0)* (deriv '(* x y) 'x) *(+ (* x 0) (* 1 y))* (deriv '(* (* x y) (+ x 3)) 'x) *(+ (* (* x y) (+ 1 0)) (* (+ (* x 0) (* 1 y)) (+ x 3)))*

El programa produce respuestas que son correctas; sin embargo, no están simplificadas. Es cierto que

$${{\it d\,(xy\,)} \over {\it dx}} = x \cdot 0 + 1 \cdot y,$$

pero nos gustaría que el programa supiera que $x \cdot 0 = 0$, $1 \cdot y = y$ y $0 + y = y$. La respuesta para el segundo ejemplo debería haber sido simplemente `y`. Como muestra el tercer ejemplo, esto se convierte en un problema serio cuando las expresiones son complejas.

Nuestra dificultad es muy parecida a la que encontramos con la implementación de los números racionales: no hemos reducido las respuestas a su forma más simple. Para lograr la reducción de los números racionales, necesitábamos cambiar solo los constructores y los selectores de la implementación. Podemos adoptar una estrategia similar aquí. No cambiaremos `deriv` en absoluto. En su lugar, cambiaremos `make/sum` para que si ambos sumandos son números, `make/sum` los sume y devuelva su suma. Además, si uno de los sumandos es 0, entonces `make/sum` devolverá el otro sumando.

(define (make-sum a1 a2) (cond ((=number? a1 0) a2) ((=number? a2 0) a1) ((and (number? a1) (number? a2)) (+ a1 a2)) (else (list '+ a1 a2))))

Esto utiliza el procedimiento `=number?`, que comprueba si una expresión es igual a un número dado:

(define (=number? exp num) (and (number? exp) (= exp num)))

De manera similar, cambiaremos `make/product` para incorporar las reglas de que 0 por cualquier cosa es 0 y 1 por cualquier cosa es la cosa misma:

(define (make-product m1 m2) (cond ((or (=number? m1 0) (=number? m2 0)) 0) ((=number? m1 1) m2) ((=number? m2 1) m1) ((and (number? m1) (number? m2)) (* m1 m2)) (else (list '* m1 m2))))

Así es como funciona esta versión en nuestros tres ejemplos:

(deriv '(+ x 3) 'x) *1* (deriv '(* x y) 'x) *y* (deriv '(* (* x y) (+ x 3)) 'x) *(+ (* x y) (* y (+ x 3)))*

Aunque esto es una mejora considerable, el tercer ejemplo muestra que

todavía hay un largo camino por recorrer antes de obtener un programa que ponga las expresiones en una forma que podríamos acordar que es la "más simple". El problema de la simplificación algebraica es complejo porque, entre otras razones, una forma que puede ser la más simple para un propósito puede no serlo para otro.

Ejercicio 2.56: Muestre cómo extender el diferenciador básico para manejar más tipos de expresiones. Por ejemplo, implemente la regla de diferenciación

$$ {{\it d\,(u^n\,)} \over {\it dx}} = nu^{n-1} {{\it du} \over {\it dx}} $$

agregando una nueva cláusula al programa `deriv` y definiendo los procedimientos apropiados `exponentiation?`, `base`, `exponent` y `make/exponentiation`. (Puede usar el símbolo `**` para denotar la exponenciación). Incorpore las reglas de que cualquier cosa elevada a la potencia 0 es 1 y cualquier cosa elevada a la potencia 1 es la cosa misma.

Ejercicio 2.57: Extienda el programa de diferenciación para manejar sumas y productos de un número arbitrario de (dos o más) términos. Entonces, el último ejemplo anterior podría expresarse como

```
(deriv '(* x y (+ x 3)) 'x)
```

Intente hacer esto cambiando solo la representación de las sumas y los productos, sin cambiar el procedimiento `deriv` en absoluto. Por ejemplo, el `addend` de una suma sería el primer término, y el `augend` sería la suma del resto de los términos.

Ejercicio 2.58: Supongamos que queremos modificar el programa de diferenciación para que funcione con la notación matemática ordinaria, en la que `+` y `*` son operadores infijos en lugar de prefijos. Dado que el programa de diferenciación se define en términos de datos abstractos, podemos modificarlo para que funcione con diferentes representaciones de expresiones simplemente cambiando los predicados, selectores y constructores que definen la representación de las expresiones algebraicas en las que debe operar el diferenciador.

1. Muestre cómo hacer esto para diferenciar expresiones algebraicas presentadas en forma infija, como `(x + (3 * (x + (y + 2))))`. Para simplificar la tarea, suponga que `+` y `*` siempre toman dos argumentos y que las expresiones están completamente entre paréntesis.

2. El problema se vuelve considerablemente más difícil si permitimos la notación algebraica estándar, como `(x + 3 * (x + y + 2))`, que elimina los paréntesis innecesarios y asume que la multiplicación se realiza antes que la suma. ¿Puede diseñar predicados, selectores y constructores apropiados para esta notación de modo que nuestro programa de derivadas siga funcionando?

Ejemplo: Representando Conjuntos

En los ejemplos anteriores construimos representaciones para dos tipos de objetos de datos compuestos: números racionales y expresiones algebraicas. En uno de estos ejemplos tuvimos la opción de simplificar (reducir) las expresiones en el momento de la construcción o en el de la selección, pero aparte de eso, la elección de una representación para estas estructuras en términos de listas fue sencilla. Cuando nos ocupamos de la representación de conjuntos, la elección de una representación no es tan obvia. De hecho, hay varias representaciones posibles, y difieren significativamente entre sí de varias maneras.

Informalmente, un conjunto es simplemente una colección de objetos distintos. Para dar una definición más precisa, podemos emplear el método de la abstracción de datos. Es decir, definimos "conjunto" especificando las operaciones que se deben usar en los conjuntos. Estas son `union/set`, `intersection/set`, `element/of/set?` y `adjoin/set`. `element/of/set?` es un predicado que determina si un elemento dado es miembro de un conjunto. `adjoin/set` toma un objeto y un conjunto como argumentos y devuelve un conjunto que contiene los elementos del conjunto original y también el elemento adjunto. `union/set` calcula la unión de dos conjuntos, que es el conjunto que contiene cada elemento que aparece en cualquiera de los argumentos. `intersection/set` calcula la intersección de dos conjuntos, que es el conjunto que contiene solo los elementos que aparecen en ambos argumentos. Desde el punto de vista de la abstracción de datos, somos libres de diseñar cualquier representación que implemente estas operaciones de una manera consistente con las interpretaciones dadas anteriormente.[37]

Conjuntos como listas no ordenadas

Una forma de representar un conjunto es como una lista de sus elementos en la que ningún elemento aparece más de una vez. El conjunto vacío se representa mediante la lista vacía. En esta representación, `element/of/set?` es similar al procedimiento `memq` de Section 2.3.1. Utiliza `equal?` en lugar de `eq?` para que los elementos del conjunto no tengan que ser símbolos:

(define (element-of-set? x set) (cond ((null? set) false) ((equal? x (car set)) true) (else (element-of-set? x (cdr set)))))

Usando esto, podemos escribir `adjoin/set`. Si el objeto que se va a adjuntar ya está en el conjunto, simplemente devolvemos el conjunto. De lo contrario, usamos `cons` para agregar el objeto a la lista que representa el conjunto:

(define (adjoin-set x set) (if (element-of-set? x set) set (cons x set)))

Para `intersection/set` podemos usar una estrategia recursiva. Si sabemos cómo formar la intersección de `set2` y el `cdr` de `set1`, solo necesitamos decidir si incluir el `car` de `set1` en esto. Pero esto depende de si `(car set1)` también está en `set2`. Aquí está el procedimiento resultante:

(define (intersection-set set1 set2) (cond ((or (null? set1) (null? set2)) '()) ((element-of-set? (car set1) set2) (cons (car set1) (intersection-set (cdr set1) set2))) (else (intersection-set (cdr set1) set2))))

Al diseñar una representación, uno de los problemas con los que debemos preocuparnos es la eficiencia. Considere el número de pasos requeridos por nuestras operaciones de conjunto. Dado que todas usan `element/of/set?`, la velocidad de esta operación tiene un gran impacto en la eficiencia de la implementación del conjunto en su conjunto. Ahora, para verificar si un objeto es miembro de un conjunto, `element/of/set?` puede tener que escanear todo el conjunto. (En el peor de los casos, el objeto resulta no estar en el conjunto). Por lo tanto, si el conjunto tiene n elementos, `element/of/set?` podría tomar hasta n pasos. Por lo tanto, el número de pasos requeridos crece como $\Theta(n)$. El número de pasos requeridos por `adjoin/set`, que usa esta operación, también crece como $\Theta(n)$. Para `intersection/set`, que realiza una verificación de `element/of/set?` para cada elemento de `set1`, el número de pasos requeridos crece como el producto de los tamaños de los conjuntos involucrados, o $\Theta(n^2)$ para dos conjuntos de tamaño n. Lo mismo será cierto para `union/set`.

Ejercicio 2.59: Implemente la operación `union/set` para la representación de listas no ordenadas de conjuntos.

Ejercicio 2.60: Especificamos que un conjunto se representaría como una lista sin duplicados. Ahora supongamos que permitimos duplicados. Por ejemplo, el conjunto $\{1, 2, 3\}$ podría representarse como la lista `(2 3 2 1 3 2 2)`. Diseñe los procedimientos `element/of/set?`, `adjoin/set`, `union/set` y `intersection/set` que operen sobre esta representación. ¿Cómo se compara la eficiencia de cada uno con el procedimiento correspondiente para la representación sin duplicados? ¿Hay aplicaciones para las que usaría esta representación en lugar de la sin duplicados?

Conjuntos como listas ordenadas

Una forma de acelerar nuestras operaciones de conjunto es cambiar la representación para que los elementos del conjunto se listen en orden creciente. Para hacer esto, necesitamos alguna forma de comparar dos objetos para poder decir cuál es más grande. Por ejemplo, podríamos comparar símbolos lexicográficamente, o podríamos acordar algún método para asignar un número único a un objeto y luego comparar los elementos comparando los números correspondientes. Para mantener nuestra discusión simple, consideraremos solo el caso en que los elementos del conjunto son números, de modo que podamos comparar elementos usando `>` y `<`. Representaremos un conjunto de números listando sus elementos en orden creciente. Mientras que nuestra primera representación anterior nos permitía representar el conjunto $\{1, 3, 6, 10\}$ listando los elementos en cualquier orden, nuestra nueva representación solo permite la lista `(1 3 6 10)`.

Una ventaja del ordenamiento se muestra en `element/of/set?`: al verificar la presencia de un elemento, ya no tenemos que escanear todo el conjunto. Si llegamos a un elemento del conjunto que es más grande que el elemento que estamos buscando, entonces sabemos que el elemento no está en el conjunto:

(define (element-of-set? x set) (cond ((null? set) false) ((= x (car set)) true) ((< x (car set)) false) (else (element-of-set? x (cdr set)))))

¿Cuántos pasos ahorra esto? En el peor de los casos, el elemento que estamos buscando puede ser el más grande del conjunto, por lo que el número de pasos es el mismo que para la representación no ordenada. Por otro lado, si buscamos elementos de muchos tamaños diferentes, podemos esperar que a veces podamos dejar de buscar en un punto cercano al principio de la lista y que otras veces todavía necesitemos examinar la mayor parte de la lista. En promedio, deberíamos esperar tener que examinar aproximadamente la mitad de los elementos del conjunto. Por lo tanto, el número promedio de pasos requeridos será de aproximadamente $n/2$. Esto sigue siendo un crecimiento de $\Theta(n)$, pero nos ahorra, en promedio, un factor de 2 en el número de pasos sobre la implementación anterior.

Obtenemos una aceleración más impresionante con `intersection/set`. En la representación no ordenada, esta operación requería $\Theta(n^2)$ pasos, porque realizábamos un escaneo completo de `set2` para cada elemento de `set1`. Pero con la representación ordenada, podemos usar un método más ingenioso. Comience comparando los elementos iniciales, `x1` y `x2`, de los dos conjuntos. Si `x1` es igual a `x2`, eso da un elemento de la intersección, y el resto de la intersección es la intersección de los `cdr` de los dos conjuntos. Suponga, sin embargo, que `x1` es menor que `x2`. Dado que `x2` es el elemento

más pequeño de `set2`, podemos concluir inmediatamente que `x1` no puede aparecer en ninguna parte de `set2` y, por lo tanto, no está en la intersección. Por lo tanto, la intersección es igual a la intersección de `set2` con el `cdr` de `set1`. De manera similar, si `x2` es menor que `x1`, entonces la intersección viene dada por la intersección de `set1` con el `cdr` de `set2`. Aquí está el procedimiento:

(define (intersection-set set1 set2) (if (or (null? set1) (null? set2)) '() (let ((x1 (car set1)) (x2 (car set2))) (cond ((= x1 x2) (cons x1 (intersection-set (cdr set1) (cdr set2)))) ((< x1 x2) (intersection-set (cdr set1) set2)) ((< x2 x1) (intersection-set set1 (cdr set2)))))))

Para estimar el número de pasos requeridos por este proceso, observe que en cada paso reducimos el problema de la intersección a calcular intersecciones de conjuntos más pequeños, eliminando el primer elemento de `set1` o `set2` o ambos. Por lo tanto, el número de pasos requeridos es como máximo la suma de los tamaños de `set1` y `set2`, en lugar del producto de los tamaños como con la representación no ordenada. Esto es un crecimiento de $\Theta(n)$ en lugar de $\Theta(n^2)$, una aceleración considerable, incluso para conjuntos de tamaño moderado.

> **Ejercicio 2.61:** Dé una implementación de `adjoin/set` usando la representación ordenada. Por analogía con `element/of/set?`, muestre cómo aprovechar el orden para producir un procedimiento que requiera en promedio aproximadamente la mitad de los pasos que con la representación no ordenada.
>
> **Ejercicio 2.62:** Dé una implementación de $\Theta(n)$ de `union/set` para conjuntos representados como listas ordenadas.

Conjuntos como árboles binarios

Podemos mejorar la representación de listas ordenadas organizando los elementos del conjunto en forma de árbol. Cada nodo del árbol contiene un elemento del conjunto, llamado la "entrada" en ese nodo, y un enlace a cada uno de los otros dos nodos (posiblemente vacíos). El enlace "izquierdo" apunta a elementos más pequeños que el del nodo, y el enlace "derecho" a elementos más grandes que el del nodo. La Figura 2.16 muestra algunos árboles que representan el conjunto $\{1, 3, 5, 7, 9, 11\}$. El mismo conjunto puede representarse mediante un árbol de varias formas diferentes. Lo único que requerimos para una representación válida es que todos los elementos del subárbol izquierdo sean más pequeños que la entrada del nodo y que todos los elementos del subárbol derecho sean más grandes.

Figura 2.16: Varios árboles binarios que representan el conjunto {1, 3, 5, 7, 9, 11}.

La ventaja de la representación en árbol es la siguiente: supongamos que queremos comprobar si un número *x* está contenido en un conjunto. Comenzamos comparando *x* con la entrada en el nodo superior. Si *x* es menor que esto, sabemos que solo necesitamos buscar en el subárbol izquierdo; si *x* es mayor, solo necesitamos buscar en el subárbol derecho. Ahora, si el árbol está "equilibrado", cada uno de estos subárboles tendrá aproximadamente la mitad del tamaño del original. Por lo tanto, en un paso hemos reducido el problema de buscar en un árbol de tamaño *n* a buscar en un árbol de tamaño *n*/2. Dado que el tamaño del árbol se reduce a la mitad en cada paso, deberíamos esperar que el número de pasos necesarios para buscar en un árbol de tamaño *n* crezca como $\Theta(\log n)$.[38] Para conjuntos grandes, esto será una aceleración significativa con respecto a las representaciones anteriores.

Podemos representar árboles usando listas. Cada nodo será una lista de tres elementos: la entrada en el nodo, el subárbol izquierdo y el subárbol derecho. Un subárbol izquierdo o derecho de la lista vacía indicará que no hay ningún subárbol conectado allí. Podemos describir esta representación mediante los siguientes procedimientos:[39]

(define (entry tree) (car tree)) (define (left-branch tree) (cadr tree)) (define (right-branch tree) (caddr tree)) (define (make-tree entry left right) (list entry left right))

Ahora podemos escribir el procedimiento `element/of/set?` usando la estrategia descrita anteriormente:

(define (element-of-set? x set) (cond ((null? set) false) ((= x (entry set)) true) ((< x (entry set)) (element-of-set? x (left-branch set))) ((> x (entry set)) (element-of-set? x (right-branch set)))))

Adjuntar un elemento a un conjunto se implementa de manera similar y también requiere $\Theta(\log n)$ pasos. Para adjuntar un elemento `x`, comparamos `x` con la entrada del nodo para determinar si `x` debe agregarse a la rama derecha o a la izquierda, y habiendo adjuntado `x` a la rama apropiada, unimos esta rama recién construida con la entrada original y la otra rama. Si `x` es igual a la entrada, simplemente devolvemos el nodo. Si se nos pide que adjuntemos `x` a un árbol vacío, generamos un árbol que tiene `x` como entrada y ramas derecha e izquierda vacías. Aquí está el procedimiento:

(define (adjoin-set x set) (cond ((null? set) (make-tree x '() '())) ((= x (entry

set)) set) ((< x (entry set)) (make-tree (entry set) (adjoin-set x (left-branch set)) (right-branch set))) ((> x (entry set)) (make-tree (entry set) (left-branch set) (adjoin-set x (right-branch set))))))

La afirmación anterior de que la búsqueda en el árbol se puede realizar en un número logarítmico de pasos se basa en la suposición de que el árbol está "equilibrado", es decir, que los subárboles izquierdo y derecho de cada árbol tienen aproximadamente el mismo número de elementos, de modo que cada subárbol contiene aproximadamente la mitad de los elementos de su padre. Pero, ¿cómo podemos estar seguros de que los árboles que construimos estarán equilibrados? Incluso si comenzamos con un árbol equilibrado, agregar elementos con `adjoin/set` puede producir un resultado desequilibrado. Dado que la posición de un elemento recién agregado depende de cómo se compara el elemento con los elementos que ya están en el conjunto, podemos esperar que si agregamos elementos "al azar", el árbol tenderá a estar equilibrado en promedio. Pero esto no es una garantía. Por ejemplo, si comenzamos con un conjunto vacío y agregamos los números del 1 al 7 en secuencia, terminamos con el árbol altamente desequilibrado que se muestra en la Figura 2.17. En este árbol, todos los subárboles izquierdos están vacíos, por lo que no tiene ninguna ventaja sobre una simple lista ordenada. Una forma de resolver este problema es definir una operación que transforme un árbol arbitrario en un árbol equilibrado con los mismos elementos. Luego, podemos realizar esta transformación después de cada pocas operaciones de `adjoin/set` para mantener nuestro conjunto en equilibrio. También hay otras formas de resolver este problema, la mayoría de las cuales implican el diseño de nuevas estructuras de datos para las cuales tanto la búsqueda como la inserción se pueden realizar en $\Theta(\log n)$ pasos.[40]

Figura 2.17: Árbol desequilibrado producido al añadir del 1 al 7 en secuencia.

Ejercicio 2.63: Cada uno de los siguientes dos procedimientos convierte un árbol binario en una lista.

(define (tree->list-1 tree) (if (null? tree) '() (append (tree->list-1 (left-branch tree)) (cons (entry tree) (tree->list-1 (right-branch tree)))))) (define (tree->list-2 tree) (define (copy-to-list tree result-list) (if (null? tree) result-list (copy-to-list (left-branch tree) (cons (entry tree) (copy-to-list (right-branch tree) result-list))))) (copy-to-list tree '()))

1. ¿Producen los dos procedimientos el mismo resultado para cada

árbol? Si no es así, ¿en qué se diferencian los resultados? ¿Qué listas producen los dos procedimientos para los árboles de la Figura 2.16?

2. ¿Tienen los dos procedimientos el mismo orden de crecimiento en el número de pasos necesarios para convertir un árbol equilibrado con *n* elementos en una lista? Si no es así, ¿cuál crece más lentamente?

Ejercicio 2.64: El siguiente procedimiento `list/>tree` convierte una lista ordenada en un árbol binario equilibrado. El procedimiento auxiliar `partial/tree` toma como argumentos un entero *n* y una lista de al menos *n* elementos y construye un árbol equilibrado que contiene los primeros *n* elementos de la lista. El resultado devuelto por `partial/tree` es un par (formado con `cons`) cuyo `car` es el árbol construido y cuyo `cdr` es la lista de elementos no incluidos en el árbol.

(define (list->tree elements) (car (partial-tree elements (length elements)))) (define (partial-tree elts n) (if (= n 0) (cons '() elts) (let ((left-size (quotient (- n 1) 2))) (let ((left-result (partial-tree elts left-size))) (let ((left-tree (car left-result)) (non-left-elts (cdr left-result)) (right-size (- n (+ left-size 1)))) (let ((this-entry (car non-left-elts)) (right-result (partial-tree (cdr non-left-elts) right-size))) (let ((right-tree (car right-result)) (remaining-elts (cdr right-result))) (cons (make-tree this-entry left-tree right-tree) remaining-elts)))))))

1. Escriba un breve párrafo explicando lo más claramente posible cómo funciona `partial/tree`. Dibuje el árbol producido por `list/>tree` para la lista `(1 3 5 7 9 11)`.

2. ¿Cuál es el orden de crecimiento en el número de pasos requeridos por `list/>tree` para convertir una lista de *n* elementos?

Ejercicio 2.65: Use los resultados de los Ejercicio 2.63 y Ejercicio 2.64 para dar implementaciones de $\Theta(n)$ de `union/set` y `intersection/set` para conjuntos implementados como árboles binarios (equilibrados).[41]

Conjuntos y recuperación de información

Hemos examinado opciones para usar listas para representar conjuntos y hemos visto cómo la elección de la representación de un objeto de datos puede tener un gran impacto en el rendimiento de los programas que usan los datos. Otra razón para concentrarse en los conjuntos es que las técnicas discutidas aquí aparecen una y otra vez en aplicaciones que involucran la recuperación de información.

Considere una base de datos que contiene un gran número de registros individuales, como los archivos de personal de una empresa o las transacciones en un sistema de contabilidad. Un sistema de gestión de datos típico dedica una gran cantidad de tiempo a acceder o modificar los datos en los registros y, por lo tanto, requiere un método eficiente para acceder a los registros. Esto se hace identificando una parte de cada registro para que sirva como una *clave* de identificación. Una clave puede ser cualquier cosa que identifique de forma única el registro. Para un archivo de personal, podría ser el número de ID de un empleado. Para un sistema de contabilidad, podría ser un número de transacción. Sea cual sea la clave, cuando definimos el registro como una estructura de datos, debemos incluir un procedimiento selector `key` que recupere la clave asociada con un registro dado.

Ahora representamos la base de datos como un conjunto de registros. Para localizar el registro con una clave dada, usamos un procedimiento `lookup`, que toma como argumentos una clave y una base de datos y que devuelve el registro que tiene esa clave, o falso si no existe tal registro. `lookup` se implementa casi de la misma manera que `element/of/set?`. Por ejemplo, si el conjunto de registros se implementa como una lista no ordenada, podríamos usar

(define (lookup given-key set-of-records) (cond ((null? set-of-records) false) ((equal? given-key (key (car set-of-records))) (car set-of-records)) (else (lookup given-key (cdr set-of-records)))))

Por supuesto, hay mejores formas de representar grandes conjuntos que como listas no ordenadas. Los sistemas de recuperación de información en los que los registros deben ser "accedidos aleatoriamente" se implementan típicamente mediante un método basado en árboles, como la representación de árboles binarios discutida anteriormente. Al diseñar un sistema de este tipo, la metodología de abstracción de datos puede ser de gran ayuda. El diseñador puede crear una implementación inicial utilizando una representación simple y directa, como listas no ordenadas. Esto no será adecuado para el sistema final, pero puede ser útil para proporcionar una base de datos "rápida y sucia" con la que probar el resto del sistema. Más adelante, la representación de los datos se puede modificar para que sea más sofisticada. Si se accede a la base de datos en términos de selectores y constructores abstractos, este cambio en la representación no requerirá ningún cambio en el resto del sistema.

> **Ejercicio 2.66:** Implemente el procedimiento `lookup` para el caso en que el conjunto de registros esté estructurado como un árbol binario, ordenado por los valores numéricos de las claves.

Ejemplo: Árboles de Codificación de Huffman

Esta sección proporciona práctica en el uso de la estructura de lista y la abstracción de datos para manipular conjuntos y árboles. La aplicación es a métodos para representar datos como secuencias de unos y ceros (bits). Por ejemplo, el código estándar ASCII utilizado para representar texto en los ordenadores codifica cada carácter como una secuencia de siete bits. El uso de siete bits nos permite distinguir 2^7, o 128, caracteres diferentes posibles. En general, si queremos distinguir n símbolos diferentes, necesitaremos usar $\log_2 n$ bits por símbolo. Si todos nuestros mensajes están compuestos por los ocho símbolos A, B, C, D, E, F, G y H, podemos elegir un código con tres bits por carácter, por ejemplo

A 000 C 010 E 100 G 110 B 001 D 011 F 101 H 111

Con este código, el mensaje

BACADAEAFABBAAAGAH

se codifica como la cadena de 54 bits

001000010000011000100000101000001001000000000110000111

Los códigos como ASCII y el código de la A a la H anterior se conocen como códigos de *longitud fija*, porque representan cada símbolo en el mensaje con el mismo número de bits. A veces es ventajoso usar códigos de *longitud variable*, en los que diferentes símbolos pueden representarse con diferentes números de bits. Por ejemplo, el código Morse no usa el mismo número de puntos y rayas para cada letra del alfabeto. En particular, E, la letra más frecuente, se representa con un solo punto. En general, si nuestros mensajes son tales que algunos símbolos aparecen con mucha frecuencia y otros muy raramente, podemos codificar los datos de manera más eficiente (es decir, usando menos bits por mensaje) si asignamos códigos más cortos a los símbolos frecuentes. Considere el siguiente código alternativo para las letras de la A a la H:

A 0 C 1010 E 1100 G 1110 B 100 D 1011 F 1101 H 1111

Con este código, el mismo mensaje anterior se codifica como la cadena

100010100101101100011010100100000111001111

Esta cadena contiene 42 bits, por lo que ahorra más del 20% en espacio en comparación con el código de longitud fija que se muestra arriba.

Una de las dificultades de usar un código de longitud variable es saber cuándo ha llegado al final de un símbolo al leer una secuencia de ceros y unos. El código Morse resuelve este problema usando un *código separador* especial (en este caso, una pausa) después de la secuencia de puntos y rayas de cada letra. Otra solución es diseñar el código de tal manera que ningún código completo para ningún símbolo sea el comienzo (o *prefijo*) del código para otro símbolo. Dicho código se llama *código de prefijo*. En el ejemplo anterior, A se codifica con 0 y B se codifica con 100, por lo que ningún otro símbolo puede tener un código que comience con 0 o con 100.

En general, podemos lograr ahorros significativos si usamos códigos de prefijo de longitud variable que aprovechen las frecuencias relativas de los símbolos en los mensajes a codificar. Un esquema particular para hacer esto se llama el método de codificación de Huffman, en honor a su descubridor, David Huffman. Un código de Huffman se puede representar como un árbol binario cuyas hojas son los símbolos que se codifican. En cada nodo no hoja del árbol hay un conjunto que contiene todos los símbolos en las hojas que se encuentran debajo del nodo. Además, a cada símbolo en una hoja se le asigna un peso (que es su frecuencia relativa), y cada nodo no hoja contiene un peso que es la suma de todos los pesos de las hojas que se encuentran debajo de él. Los pesos no se usan en el proceso de codificación o decodificación. Veremos a continuación cómo se usan para ayudar a construir el árbol.

La Figura 2.18 muestra el árbol de Huffman para el código de la A a la H dado anteriormente. Los pesos en las hojas indican que el árbol fue diseñado para mensajes en los que A aparece con una frecuencia relativa de 8, B con una frecuencia relativa de 3, y las otras letras cada una con una frecuencia relativa de 1.

Figura 2.18: Un árbol de codificación de Huffman.

Dado un árbol de Huffman, podemos encontrar la codificación de cualquier símbolo comenzando en la raíz y descendiendo hasta llegar a la hoja que contiene el símbolo. Cada vez que descendemos por una rama izquierda, añadimos un 0 al código, y cada vez que descendemos por una rama derecha, añadimos un 1. (Decidimos qué rama seguir probando para ver qué rama es el nodo hoja para el símbolo o contiene el símbolo en su conjunto). Por ejemplo, comenzando desde la raíz del árbol en la Figura 2.18, llegamos a la hoja para D siguiendo una rama derecha, luego una rama izquierda, luego una rama derecha, luego una rama derecha; por lo tanto, el código para D es 1011.

Para decodificar una secuencia de bits usando un árbol de Huffman, comenzamos en la raíz y usamos los sucesivos ceros y unos de la secuencia de bits para determinar si debemos descender por la rama izquierda o por la derecha. Cada vez que llegamos a una hoja, hemos generado un nuevo símbolo en el mensaje, momento en el cual comenzamos de nuevo desde la raíz del árbol para encontrar el siguiente símbolo. Por ejemplo, supongamos que se nos da el árbol anterior y la secuencia 10001010. Comenzando en la raíz, descendemos por la rama derecha (ya que el primer bit de la cadena es 1), luego por la rama izquierda (ya que el segundo bit es 0), luego por la rama izquierda (ya que el tercer bit también es 0). Esto nos lleva a la hoja de B, por lo que el primer símbolo del mensaje decodificado es B. Ahora comenzamos de nuevo en la raíz, y hacemos un movimiento a la izquierda porque el siguiente bit de la cadena es 0. Esto nos lleva a la hoja de A. Luego comenzamos de nuevo en la raíz con el resto de la cadena 1010, por lo que nos movemos a la derecha, a la izquierda, a la derecha, a la izquierda y llegamos a C. Por lo tanto, el mensaje completo es BAC.

Generando árboles de Huffman

Dado un "alfabeto" de símbolos y sus frecuencias relativas, ¿cómo construimos el "mejor" código? (En otras palabras, ¿qué árbol codificará los mensajes con la menor cantidad de bits?) Huffman dio un algoritmo para hacer esto y demostró que el código resultante es de hecho el mejor código de longitud variable para mensajes donde la frecuencia relativa de los símbolos coincide con las frecuencias con las que se construyó el código. No demostraremos esta optimalidad de los códigos de Huffman aquí, pero mostraremos cómo se construyen los árboles de Huffman.[42]

El algoritmo para generar un árbol de Huffman es muy simple. La idea es organizar el árbol de modo que los símbolos con la frecuencia más baja aparezcan más lejos de la raíz. Comience con el conjunto de nodos hoja, que contienen los símbolos y sus frecuencias, según lo determinado por los datos iniciales a partir de los cuales se construirá el código. Ahora encuentre dos hojas con los pesos más bajos y combínelas para producir un nodo que tenga estos dos nodos como sus ramas izquierda y derecha. El peso del nuevo nodo es la suma de los dos pesos. Elimine las dos hojas del conjunto original y reemplácelas por este nuevo nodo. Ahora continúe este proceso. En cada paso, combine dos nodos con los pesos más pequeños, eliminándolos del conjunto y reemplazándolos por un nodo que tenga estos dos como sus ramas izquierda y derecha. El proceso se detiene cuando solo queda un nodo, que es la raíz de todo el árbol. Así es como se generó el árbol de Huffman de la Figura 2.18:

Hojas iniciales (A 8) (B 3) (C 1) (D 1) (E 1) (F 1) (G 1) (H 1) Fusionar (A 8)

(B 3) (C D 2) (E 1) (F 1) (G 1) (H 1) Fusionar (A 8) (B 3) (C D 2) (E F 2) (G 1) (H 1) Fusionar (A 8) (B 3) (C D 2) (E F 2) (G H 2) Fusionar (A 8) (B 3) (C D 2) (E F G H 4) Fusionar (A 8) (B C D 5) (E F G H 4) Fusionar (A 8) (B C D E F G H 9) Fusión final (A B C D E F G H 17)

El algoritmo no siempre especifica un árbol único, porque puede que no haya nodos de peso más bajo únicos en cada paso. Además, la elección del orden en que se fusionan los dos nodos (es decir, cuál será la rama derecha y cuál será la rama izquierda) es arbitraria.

Representación de los árboles de Huffman

En los ejercicios siguientes trabajaremos con un sistema que utiliza árboles de Huffman para codificar y decodificar mensajes y genera árboles de Huffman según el algoritmo descrito anteriormente. Comenzaremos discutiendo cómo se representan los árboles.

Las hojas del árbol se representan mediante una lista que consiste en el símbolo `leaf`, el símbolo en la hoja y el peso:

```
(define (make-leaf symbol weight) (list 'leaf symbol weight)) (define (leaf? object) (eq? (car object) 'leaf)) (define (symbol-leaf x) (cadr x)) (define (weight-leaf x) (caddr x))
```

Un árbol general será una lista de una rama izquierda, una rama derecha, un conjunto de símbolos y un peso. El conjunto de símbolos será simplemente una lista de los símbolos, en lugar de una representación de conjunto más sofisticada. Cuando creamos un árbol fusionando dos nodos, obtenemos el peso del árbol como la suma de los pesos de los nodos, y el conjunto de símbolos como la unión de los conjuntos de símbolos de los nodos. Dado que nuestros conjuntos de símbolos se representan como listas, podemos formar la unión usando el procedimiento `append` que definimos en Section 2.2.1:

```
(define (make-code-tree left right) (list left right (append (symbols left) (symbols right)) ( + (weight left) (weight right))))
```

Si creamos un árbol de esta manera, tenemos los siguientes selectores:

```
(define (left-branch tree) (car tree)) (define (right-branch tree) (cadr tree)) (define (symbols tree) (if (leaf? tree) (list (symbol-leaf tree)) (caddr tree))) (define (weight tree) (if (leaf? tree) (weight-leaf tree) (cadddr tree)))
```

Los procedimientos `symbols` y `weight` deben hacer algo ligeramente diferente dependiendo de si se les llama con una hoja o con un árbol general. Estos son ejemplos simples de *procedimientos genéricos* (procedimientos que

pueden manejar más de un tipo de datos), sobre los cuales tendremos mucho más que decir en Section 2.4 y Section 2.5.

El procedimiento de decodificación

El siguiente procedimiento implementa el algoritmo de decodificación. Toma como argumentos una lista de ceros y unos, junto con un árbol de Huffman.

(define (decode bits tree) (define (decode-1 bits current-branch) (if (null? bits) '() (let ((next-branch (choose-branch (car bits) current-branch))) (if (leaf? next-branch) (cons (symbol-leaf next-branch) (decode-1 (cdr bits) tree)) (decode-1 (cdr bits) next-branch))))) (decode-1 bits tree)) (define (choose-branch bit branch) (cond ((= bit 0) (left-branch branch)) ((= bit 1) (right-branch branch)) (else (error "bad bit: CHOOSE-BRANCH" bit))))

El procedimiento `decode/1` toma dos argumentos: la lista de bits restantes y la posición actual en el árbol. Continúa moviéndose "hacia abajo" en el árbol, eligiendo una rama izquierda o derecha según si el siguiente bit de la lista es un cero o un uno. (Esto se hace con el procedimiento `choose/branch`.) Cuando llega a una hoja, devuelve el símbolo de esa hoja como el siguiente símbolo del mensaje, añadiéndolo con `cons` al resultado de decodificar el resto del mensaje, comenzando desde la raíz del árbol. Observe la comprobación de errores en la última cláusula de `choose/branch`, que se queja si el procedimiento encuentra algo que no sea un cero o un uno en los datos de entrada.

Conjuntos de elementos ponderados

En nuestra representación de árboles, cada nodo no hoja contiene un conjunto de símbolos, que hemos representado como una lista simple. Sin embargo, el algoritmo de generación de árboles discutido anteriormente requiere que también trabajemos con conjuntos de hojas y árboles, fusionando sucesivamente los dos elementos más pequeños. Dado que se nos pedirá que encontremos repetidamente el elemento más pequeño en un conjunto, es conveniente usar una representación ordenada para este tipo de conjunto.

Representaremos un conjunto de hojas y árboles como una lista de elementos, dispuestos en orden creciente de peso. El siguiente procedimiento `adjoin/set` para construir conjuntos es similar al descrito en el Ejercicio 2.61; sin embargo, los elementos se comparan por sus pesos, y el elemento que se agrega al conjunto nunca está ya en él.

(define (adjoin-set x set) (cond ((null? set) (list x)) ((< (weight x) (weight

(car set))) (cons x set)) (else (cons (car set) (adjoin-set x (cdr set))))))

El siguiente procedimiento toma una lista de pares símbolo-frecuencia como `((A 4) (B 2) (C 1) (D 1))` y construye un conjunto ordenado inicial de hojas, listo para ser fusionado según el algoritmo de Huffman:

(define (make-leaf-set pairs) (if (null? pairs) '() (let ((pair (car pairs))) (adjoin-set (make-leaf (car pair) ; símbolo (cadr pair)) ; frecuencia (make-leaf-set (cdr pairs))))))

Ejercicio 2.67: Defina un árbol de codificación y un mensaje de muestra:

(define sample-tree (make-code-tree (make-leaf 'A 4) (make-code-tree (make-leaf 'B 2) (make-code-tree (make-leaf 'D 1) (make-leaf 'C 1))))) (define sample-message '(0 1 1 0 0 1 0 1 0 1 1 1 0))

Use el procedimiento `decode` para decodificar el mensaje, y dé el resultado.

Ejercicio 2.68: El procedimiento `encode` toma como argumentos un mensaje y un árbol y produce la lista de bits que da el mensaje codificado.

(define (encode message tree) (if (null? message) '() (append (encode-symbol (car message) tree) (encode (cdr message) tree))))

`encode/symbol` es un procedimiento, que debe escribir, que devuelve la lista de bits que codifica un símbolo dado según un árbol dado. Debe diseñar `encode/symbol` para que señale un error si el símbolo no está en el árbol en absoluto. Pruebe su procedimiento codificando el resultado que obtuvo en el Ejercicio 2.67 con el árbol de muestra y viendo si es el mismo que el mensaje de muestra original.

Ejercicio 2.69: El siguiente procedimiento toma como argumento una lista de pares símbolo-frecuencia (donde ningún símbolo aparece en más de un par) y genera un árbol de codificación de Huffman según el algoritmo de Huffman.

(define (generate-huffman-tree pairs) (successive-merge (make-leaf-set pairs)))

`make/leaf/set` es el procedimiento dado anteriormente que transforma la lista de pares en un conjunto ordenado de hojas. `successive/merge` es el procedimiento que debe escribir, usando `make/code/tree` para fusionar sucesivamente los elementos de menor peso del conjunto hasta que solo

quede un elemento, que es el árbol de Huffman deseado. (Este procedimiento es un poco complicado, pero no realmente complejo. Si se encuentra diseñando un procedimiento complejo, entonces es casi seguro que está haciendo algo mal. Puede aprovechar significativamente el hecho de que estamos usando una representación de conjunto ordenado).

Ejercicio 2.70: El siguiente alfabeto de ocho símbolos con frecuencias relativas asociadas fue diseñado para codificar eficientemente las letras de las canciones de rock de los años 50. (Tenga en cuenta que los "símbolos" de un "alfabeto" no tienen por qué ser letras individuales).

A 2 GET 2 SHA 3 WAH 1 BOOM 1 JOB 2 NA 16 YIP 9

Use `generate/huffman/tree` (Ejercicio 2.69) para generar un árbol de Huffman correspondiente, y use `encode` (Ejercicio 2.68) para codificar el siguiente mensaje:

Get a job Sha na na na na na na na na Get a job Sha na na na na na na na na Wah yip yip yip yip yip yip yip yip yip Sha boom

¿Cuántos bits se requieren para la codificación? ¿Cuál es el número más pequeño de bits que se necesitarían para codificar esta canción si usáramos un código de longitud fija para el alfabeto de ocho símbolos?

Ejercicio 2.71: Supongamos que tenemos un árbol de Huffman para un alfabeto de n símbolos, y que las frecuencias relativas de los símbolos son $1, 2, 4, \ldots, 2^{n-1}$. Dibuja el árbol para $n = 5$; para $n = 10$. En tal árbol (para un n general) ¿cuántos bits se requieren para codificar el símbolo más frecuente? ¿Y el símbolo menos frecuente?

Ejercicio 2.72: Considera el procedimiento de codificación que diseñaste en el Ejercicio 2.68. ¿Cuál es el orden de crecimiento en el número de pasos necesarios para codificar un símbolo? Asegúrate de incluir el número de pasos necesarios para buscar en la lista de símbolos en cada nodo encontrado. Responder a esta pregunta en general es difícil. Considera el caso especial donde las frecuencias relativas de los n símbolos son como se describe en el Ejercicio 2.71, y da el orden de crecimiento (como una función de n) del número de pasos necesarios para codificar los símbolos más y menos frecuentes en el alfabeto.

Múltiples Representaciones para Datos Abstractos

Hemos introducido la abstracción de datos, una metodología para estructurar sistemas de tal manera que gran parte de un programa pueda ser especificado independientemente de las decisiones involucradas en la implementación de los objetos de datos que el programa manipula. Por ejemplo, vimos en Section 2.1.1 cómo separar la tarea de diseñar un programa que usa números racionales de la tarea de implementar números racionales en términos de los mecanismos primitivos del lenguaje de la computadora para construir datos compuestos. La idea clave fue erigir una barrera de abstracción —en este caso, los selectores y constructores para números racionales (`make/rat`, `numer`, `denom`)— que aísla la forma en que se utilizan los números racionales de su representación subyacente en términos de estructura de lista. Una barrera de abstracción similar aísla los detalles de los procedimientos que realizan aritmética racional (`add/rat`, `sub/rat`, `mul/rat` y `div/rat`) de los procedimientos de "nivel superior" que usan números racionales. El programa resultante tiene la estructura que se muestra en la Figura 2.1.

Estas barreras de abstracción de datos son herramientas poderosas para controlar la complejidad. Al aislar las representaciones subyacentes de los objetos de datos, podemos dividir la tarea de diseñar un programa grande en tareas más pequeñas que se pueden realizar por separado. Pero este tipo de abstracción de datos todavía no es lo suficientemente potente, porque puede que no siempre tenga sentido hablar de "la representación subyacente" para un objeto de datos.

Por un lado, podría haber más de una representación útil para un objeto de datos, y nos gustaría diseñar sistemas que puedan manejar múltiples representaciones. Para tomar un ejemplo simple, los números complejos pueden ser representados de dos maneras casi equivalentes: en forma rectangular (partes real e imaginaria) y en forma polar (magnitud y ángulo). A veces la forma rectangular es más apropiada y a veces la forma polar es más apropiada. De hecho, es perfectamente plausible imaginar un sistema en el que los números complejos se representen de ambas maneras, y en el que los procedimientos para manipular números complejos funcionen con cualquiera de las representaciones.

Más importante aún, los sistemas de programación a menudo son diseñados por muchas personas que trabajan durante largos períodos de tiempo, sujetos a requisitos que cambian con el tiempo. En tal entorno, simplemente no es posible que todos se pongan de acuerdo de antemano sobre las elecciones de representación de datos. Así que además de las barreras de abstracción de datos que aíslan la representación del uso, necesitamos barreras de abstracción que aíslen diferentes opciones de diseño entre sí y permitan que diferentes opciones coexistan en un solo programa. Además, dado que los programas grandes a menudo se crean combinando módulos preexistentes

que fueron diseñados de forma aislada, necesitamos convenciones que permitan a los programadores incorporar módulos en sistemas más grandes de forma *aditiva*, es decir, sin tener que rediseñar o reimplementar estos módulos.

En esta sección, aprenderemos cómo lidiar con datos que pueden ser representados de diferentes maneras por diferentes partes de un programa. Esto requiere construir *procedimientos genéricos*—procedimientos que pueden operar sobre datos que pueden ser representados en más de una forma. Nuestra técnica principal para construir procedimientos genéricos será trabajar en términos de objetos de datos que tienen *etiquetas de tipo*, es decir, objetos de datos que incluyen información explícita sobre cómo deben ser procesados. También discutiremos la programación *dirigida por datos*, una estrategia de implementación potente y conveniente para ensamblar aditivamente sistemas con operaciones genéricas.

Comenzamos con el ejemplo simple de números complejos. Veremos cómo las etiquetas de tipo y el estilo dirigido por datos nos permiten diseñar representaciones rectangulares y polares separadas para los números complejos mientras mantenemos la noción de un objeto de datos abstracto de "número complejo". Lograremos esto definiendo procedimientos aritméticos para números complejos (`add/complex`, `sub/complex`, `mul/complex` y `div/complex`) en términos de selectores genéricos que acceden a partes de un número complejo independientemente de cómo se represente el número. El sistema de números complejos resultante, como se muestra en la Figura 2.19, contiene dos tipos diferentes de barreras de abstracción. Las barreras de abstracción "horizontales" juegan el mismo papel que las de la Figura 2.1. Aíslan las operaciones de "nivel superior" de las representaciones de "nivel inferior". Además, hay una barrera "vertical" que nos da la capacidad de diseñar e instalar por separado representaciones alternativas.

Figura 2.19: Barreras de abstracción de datos en el sistema de números complejos.

En Section 2.5 mostraremos cómo usar etiquetas de tipo y estilo dirigido por datos para desarrollar un paquete de aritmética genérica. Esto proporciona procedimientos (`add`, `mul`, etc.) que pueden ser usados para manipular todo tipo de "números" y pueden ser fácilmente extendidos cuando se necesita un nuevo tipo de número. En Section 2.5.3, mostraremos cómo usar la aritmética genérica en un sistema que realiza álgebra simbólica.

Representaciones para Números Complejos

Desarrollaremos un sistema que realiza operaciones aritméticas sobre números complejos como un ejemplo simple pero poco realista de un programa que utiliza operaciones genéricas. Comenzamos discutiendo dos representaciones plausibles para los números complejos como pares ordenados: forma rectangular (parte real y parte imaginaria) y forma polar (magnitud y ángulo).[43] Section 2.4.2 mostrará cómo ambas representaciones pueden coexistir en un único sistema mediante el uso de etiquetas de tipo y operaciones genéricas.

Al igual que los números racionales, los números complejos se representan naturalmente como pares ordenados. El conjunto de números complejos puede ser pensado como un espacio bidimensional con dos ejes ortogonales, el eje "real" y el eje "imaginario". (Ver Figura 2.20.) Desde este punto de vista, el número complejo $z = x + iy$ (donde $i^2 = -1$) puede ser pensado como el punto en el plano cuya coordenada real es x y cuya coordenada imaginaria es y. La adición de números complejos se reduce en esta representación a la adición de coordenadas:

$$\begin{array}{r@{{}={}}l} \hbox{Parte-real} (z_1 + z_2)\; & \hbox{ Parte-real} (z_1)\; + \hbox{ Parte-real} (z_2), \\ \hbox{Parte-imaginaria} (z_1 + z_2)\; & \hbox{ Parte-imaginaria} (z_1)\; + \hbox{ Parte-imaginaria} (z_2). \end{array}$$

Figura 2.20: Números complejos como puntos en el plano.

Al multiplicar números complejos, es más natural pensar en términos de representar un número complejo en forma polar, como una magnitud y un ángulo (r y A en la Figura 2.20). El producto de dos números complejos es el vector obtenido al estirar un número complejo por la longitud del otro y luego rotarlo a través del ángulo del otro:

$$\begin{array}{r@{{}={}}l} \hbox{Magnitud} (z_1 \cdot z_2)\; & \hbox{ Magnitud} (z_1)\; \cdot \hbox{ Magnitud} (z_2), \\ \hbox{Angulo} (z_1 \cdot z_2)\; & \hbox{ Angulo} (z_1)\; + \hbox{ Angulo} (z_2). \end{array}$$

Así, hay dos representaciones diferentes para los números complejos, que son apropiadas para diferentes operaciones. Sin embargo, desde el punto de vista de alguien que escribe un programa que utiliza números complejos, el principio de abstracción de datos sugiere que todas las operaciones para manipular números complejos deberían estar disponibles independientemente de qué representación se use por la computadora. Por ejemplo, a menudo es útil poder encontrar la magnitud de un número

complejo que se especifica por coordenadas rectangulares. Del mismo modo, a menudo es útil poder determinar la parte real de un número complejo que se especifica por coordenadas polares.

Para diseñar tal sistema, podemos seguir la misma estrategia de abstracción de datos que seguimos al diseñar el paquete de números racionales en Section 2.1.1. Asumamos que las operaciones sobre números complejos se implementan en términos de cuatro selectores: `real/part`, `imag/part`, `magnitude` y `angle`. También asumamos que tenemos dos procedimientos para construir números complejos: `make/from/real/imag` devuelve un número complejo con partes real e imaginaria especificadas, y `make/from/mag/ang` devuelve un número complejo con magnitud y ángulo especificados. Estos procedimientos tienen la propiedad de que, para cualquier número complejo `z`, tanto

(make-from-real-imag (real-part z) (imag-part z))

como

(make-from-mag-ang (magnitude z) (angle z))

producen números complejos que son iguales a `z`.

Usando estos constructores y selectores, podemos implementar la aritmética en números complejos usando los "datos abstractos" especificados por los constructores y selectores, tal como lo hicimos para los números racionales en Section 2.1.1. Como se muestra en las fórmulas anteriores, podemos sumar y restar números complejos en términos de partes real e imaginaria mientras que multiplicamos y dividimos números complejos en términos de magnitudes y ángulos:

(define (add-complex z1 z2) (make-from-real-imag (+ (real-part z1) (real-part z2)) (+ (imag-part z1) (imag-part z2)))) (define (sub-complex z1 z2) (make-from-real-imag (- (real-part z1) (real-part z2)) (- (imag-part z1) (imag-part z2)))) (define (mul-complex z1 z2) (make-from-mag-ang (* (magnitude z1) (magnitude z2)) (+ (angle z1) (angle z2)))) (define (div-complex z1 z2) (make-from-mag-ang (/ (magnitude z1) (magnitude z2)) (- (angle z1) (angle z2))))

Para completar el paquete de números complejos, debemos elegir una representación y debemos implementar los constructores y selectores en términos de números primitivos y estructura de lista primitiva. Hay dos maneras obvias de hacer esto: Podemos representar un número complejo en "forma rectangular" como un par (parte real, parte imaginaria) o en "forma polar" como un par (magnitud, ángulo). ¿Cuál debemos elegir?

Para hacer concretas las diferentes opciones, imaginemos que hay dos programadores, Ben Bitdiddle y Alyssa P. Hacker, que están diseñando de forma independiente representaciones para el sistema de números complejos. Ben elige representar los números complejos en forma rectangular. Con esta elección, seleccionar las partes real e imaginaria de un número complejo es sencillo, como lo es construir un número complejo con partes real e imaginaria dadas. Para encontrar la magnitud y el ángulo, o para construir un número complejo con una magnitud y ángulo dados, él utiliza las relaciones trigonométricas

$$\begin{array}{r@{{}={}}lr@{{}={}}l} x & r \cos A, \qquad & r & \sqrt{x^2 + y^2}, \\ y & r \sin A, \qquad & A & \arctan(y, x), \end{array}$$

que relacionan las partes real e imaginaria (x, y) con la magnitud y el ángulo (r, A).[44] La representación de Ben es por lo tanto dada por los siguientes selectores y constructores:

```
(define (real-part z) (car z)) (define (imag-part z) (cdr z)) (define (magnitude z) (sqrt (+ (square (real-part z)) (square (imag-part z))))) (define (angle z) (atan (imag-part z) (real-part z))) (define (make-from-real-imag x y) (cons x y)) (define (make-from-mag-ang r a) (cons (* r (cos a)) (* r (sin a))))
```

Alyssa, en cambio, elige representar los números complejos en forma polar. Para ella, seleccionar la magnitud y el ángulo es sencillo, pero tiene que usar las relaciones trigonométricas para obtener las partes real e imaginaria. La representación de Alyssa es:

```
(define (real-part z) (* (magnitude z) (cos (angle z)))) (define (imag-part z) (* (magnitude z) (sin (angle z)))) (define (magnitude z) (car z)) (define (angle z) (cdr z)) (define (make-from-real-imag x y) (cons (sqrt (+ (square x) (square y))) (atan y x))) (define (make-from-mag-ang r a) (cons r a))
```

La disciplina de la abstracción de datos asegura que la misma implementación de `add/complex`, `sub/complex`, `mul/complex` y `div/complex` funcionará con la representación de Ben o con la representación de Alyssa.

Datos Etiquetados

Una forma de ver la abstracción de datos es como una aplicación del "principio del menor compromiso". Al implementar el sistema de números complejos en Section 2.4.1, podemos usar la representación rectangular de Ben o la representación polar de Alyssa. La barrera de abstracción formada por los selectores y constructores nos permite aplazar hasta el último momento posible la elección de una representación concreta para nuestros

objetos de datos y así retener la máxima flexibilidad en el diseño de nuestro sistema.

El principio del menor compromiso puede llevarse a extremos aún mayores. Si lo deseamos, podemos mantener la ambigüedad de la representación incluso *después* de haber diseñado los selectores y constructores, y elegir usar tanto la representación de Ben *como* la representación de Alyssa. Sin embargo, si ambas representaciones se incluyen en un solo sistema, necesitaremos alguna forma de distinguir los datos en forma polar de los datos en forma rectangular. De lo contrario, si se nos pidiera, por ejemplo, encontrar la `magnitud` del par (3, 4), no sabríamos si responder 5 (interpretando el número en forma rectangular) o 3 (interpretando el número en forma polar). Una forma sencilla de lograr esta distinción es incluir una *etiqueta de tipo*—el símbolo `rectangular` o `polar`—como parte de cada número complejo. Luego, cuando necesitemos manipular un número complejo, podemos usar la etiqueta para decidir qué selector aplicar.

Para manipular datos etiquetados, asumiremos que tenemos procedimientos `type/tag` y `contents` que extraen de un objeto de datos la etiqueta y el contenido real (las coordenadas polares o rectangulares, en el caso de un número complejo). También postularemos un procedimiento `attach/tag` que toma una etiqueta y contenidos y produce un objeto de datos etiquetado. Una forma sencilla de implementar esto es usar la estructura de lista ordinaria:

(define (attach-tag type-tag contents) (cons type-tag contents)) (define (type-tag datum) (if (pair? datum) (car datum) (error "Mal dato etiquetado: TYPE-TAG" datum))) (define (contents datum) (if (pair? datum) (cdr datum) (error "Mal dato etiquetado: CONTENTS" datum)))

Usando estos procedimientos, podemos definir los predicados `rectangular?` y `polar?`, que reconocen números rectangulares y polares, respectivamente:

(define (rectangular? z) (eq? (type-tag z) 'rectangular)) (define (polar? z) (eq? (type-tag z) 'polar))

Con etiquetas de tipo, Ben y Alyssa ahora pueden modificar su código para que sus dos diferentes representaciones puedan coexistir en el mismo sistema. Cada vez que Ben construye un número complejo, lo etiqueta como rectangular. Cada vez que Alyssa construye un número complejo, lo etiqueta como polar. Además, Ben y Alyssa deben asegurarse de que los nombres de sus procedimientos no entren en conflicto. Una forma de hacer esto es que Ben agregue el sufijo `rectangular` al nombre de cada uno de sus procedimientos de representación y que Alyssa agregue `polar` a los nombres de los suyos. Aquí está la representación rectangular revisada de Ben de

Section 2.4.1:

(define (real-part-rectangular z) (car z)) (define (imag-part-rectangular z) (cdr z)) (define (magnitude-rectangular z) (sqrt (+ (square (real-part-rectangular z)) (square (imag-part-rectangular z))))) (define (angle-rectangular z) (atan (imag-part-rectangular z) (real-part-rectangular z))) (define (make-from-real-imag-rectangular x y) (attach-tag 'rectangular (cons x y))) (define (make-from-mag-ang-rectangular r a) (attach-tag 'rectangular (cons (* r (cos a)) (* r (sin a)))))

y aquí está la representación polar revisada de Alyssa:

(define (real-part-polar z) (* (magnitude-polar z) (cos (angle-polar z)))) (define (imag-part-polar z) (* (magnitude-polar z) (sin (angle-polar z)))) (define (magnitude-polar z) (car z)) (define (angle-polar z) (cdr z)) (define (make-from-real-imag-polar x y) (attach-tag 'polar (cons (sqrt (+ (square x) (square y))) (atan y x)))) (define (make-from-mag-ang-polar r a) (attach-tag 'polar (cons r a)))

Cada selector genérico se implementa como un procedimiento que verifica la etiqueta de su argumento y llama al procedimiento apropiado para manejar datos de ese tipo. Por ejemplo, para obtener la parte real de un número complejo, `real/part` examina la etiqueta para determinar si usar el `real/part/rectangular` de Ben o el `real/part/polar` de Alyssa. En cualquier caso, usamos `contents` para extraer el dato desnudo, sin etiqueta, y enviarlo al procedimiento rectangular o polar según sea necesario:

(define (real-part z) (cond ((rectangular? z) (real-part-rectangular (contents z))) ((polar? z) (real-part-polar (contents z))) (else (error "Tipo desconocido: REAL-PART" z)))) (define (imag-part z) (cond ((rectangular? z) (imag-part-rectangular (contents z))) ((polar? z) (imag-part-polar (contents z))) (else (error "Tipo desconocido: IMAG-PART" z)))) (define (magnitude z) (cond ((rectangular? z) (magnitude-rectangular (contents z))) ((polar? z) (magnitude-polar (contents z))) (else (error "Tipo desconocido: MAGNITUDE" z)))) (define (angle z) (cond ((rectangular? z) (angle-rectangular (contents z))) ((polar? z) (angle-polar (contents z))) (else (error "Tipo desconocido: ANGLE" z))))

Para implementar las operaciones aritméticas de números complejos, podemos usar los mismos procedimientos `add/complex`, `sub/complex`, `mul/complex` y `div/complex` de Section 2.4.1, porque los selectores que llaman son genéricos y, por lo tanto, funcionarán con cualquier representación. Por ejemplo, el procedimiento `add/complex` sigue siendo

(define (add-complex z1 z2) (make-from-real-imag (+ (real-part z1) (real-

part z2)) (+ (imag-part z1) (imag-part z2))))

Finalmente, debemos elegir si construir números complejos usando la representación de Ben o la representación de Alyssa. Una elección razonable es construir números rectangulares siempre que tengamos partes reales e imaginarias y construir números polares siempre que tengamos magnitudes y ángulos:

(define (make-from-real-imag x y) (make-from-real-imag-rectangular x y))
(define (make-from-mag-ang r a) (make-from-mag-ang-polar r a))

El sistema de números complejos resultante tiene la estructura que se muestra en la Figura 2.21. El sistema se ha descompuesto en tres partes relativamente independientes: las operaciones aritméticas de números complejos, la implementación polar de Alyssa y la implementación rectangular de Ben. Las implementaciones polar y rectangular podrían haber sido escritas por Ben y Alyssa trabajando por separado, y ambas pueden ser utilizadas como representaciones subyacentes por un tercer programador que implementa los procedimientos de aritmética compleja en términos de la interfaz abstracta de constructores/selectores.

Figura 2.21: Estructura del sistema de aritmética compleja genérica.

Dado que cada objeto de datos está etiquetado con su tipo, los selectores operan sobre los datos de manera genérica. Es decir, cada selector se define para tener un comportamiento que depende del tipo particular de dato al que se aplica. Nótese el mecanismo general para interconectar las representaciones separadas: dentro de una implementación de representación dada (digamos, el paquete polar de Alyssa) un número complejo es un par sin tipo (magnitud, ángulo). Cuando un selector genérico opera sobre un número de tipo `polar`, quita la etiqueta y pasa el contenido al código de Alyssa. A la inversa, cuando Alyssa construye un número para uso general, lo etiqueta con un tipo para que pueda ser reconocido apropiadamente por los procedimientos de nivel superior. Esta disciplina de quitar y adjuntar etiquetas a medida que los objetos de datos pasan de un nivel a otro puede ser una importante estrategia organizativa, como veremos en Section 2.5.

Programación Dirigida por Datos y Aditividad

La estrategia general de verificar el tipo de un dato y llamar a un procedimiento apropiado se llama *despacho según el tipo*. Esta es una estrategia poderosa para obtener modularidad en el diseño de sistemas. Por

otro lado, implementar el despacho como en Section 2.4.2 tiene dos debilidades significativas. Una debilidad es que los procedimientos de interfaz genérica (`real/part`, `imag/part`, `magnitude` y `angle`) deben conocer todas las diferentes representaciones. Por ejemplo, supongamos que quisiéramos incorporar una nueva representación para los números complejos en nuestro sistema de números complejos. Necesitaríamos identificar esta nueva representación con un tipo, y luego agregar una cláusula a cada uno de los procedimientos de interfaz genérica para verificar el nuevo tipo y aplicar el selector apropiado para esa representación.

Otra debilidad de la técnica es que aunque las representaciones individuales pueden diseñarse por separado, debemos garantizar que no haya dos procedimientos en todo el sistema que tengan el mismo nombre. Es por eso que Ben y Alyssa tuvieron que cambiar los nombres de sus procedimientos originales de la Sección 2.4.1.

El problema subyacente a ambas debilidades es que la técnica para implementar interfaces genéricas no es *aditiva*. La persona que implementa los procedimientos de selector genérico debe modificar esos procedimientos cada vez que se instala una nueva representación, y las personas que interconectan las representaciones individuales deben modificar su código para evitar conflictos de nombres. En cada uno de estos casos, los cambios que deben hacerse en el código son sencillos, pero deben hacerse de todos modos, y esto es una fuente de inconveniencia y error. Esto no es un gran problema para el sistema de números complejos tal como está, pero supongamos que no hubiera dos sino cientos de representaciones diferentes para los números complejos. Y supongamos que hubiera muchos selectores genéricos que mantener en la interfaz de datos abstractos. Supongamos, de hecho, que ningún programador conociera todos los procedimientos de la interfaz o todas las representaciones. El problema es real y debe abordarse en programas como los sistemas de gestión de bases de datos a gran escala.

Lo que necesitamos es un medio para modularizar aún más el diseño del sistema. Esto lo proporciona la técnica de programación conocida como *programación dirigida por datos*. Para entender cómo funciona la programación dirigida por datos, comencemos con la observación de que cada vez que tratamos con un conjunto de operaciones genéricas que son comunes a un conjunto de diferentes tipos, estamos, en efecto, tratando con una tabla bidimensional que contiene las operaciones posibles en un eje y los tipos posibles en el otro eje. Las entradas en la tabla son los procedimientos que implementan cada operación para cada tipo de argumento presentado. En el sistema de números complejos desarrollado en la sección anterior, la correspondencia entre el nombre de la operación, el tipo de dato y el procedimiento real estaba distribuida entre las diversas cláusulas

condicionales en los procedimientos de la interfaz genérica. Pero la misma información podría haberse organizado en una tabla, como se muestra en la Figura 2.22.

La programación dirigida por datos es la técnica de diseñar programas para trabajar con dicha tabla directamente. Anteriormente, implementamos el mecanismo que interconecta el código de aritmética compleja con los dos paquetes de representación como un conjunto de procedimientos que realizan cada uno un despacho explícito según el tipo. Aquí implementaremos la interfaz como un único procedimiento que busca la combinación del nombre de la operación y el tipo de argumento en la tabla para encontrar el procedimiento correcto a aplicar, y luego lo aplica al contenido del argumento. Si hacemos esto, entonces para agregar un nuevo paquete de representación al sistema no necesitamos cambiar ningún procedimiento existente; solo necesitamos agregar nuevas entradas a la tabla.

Figura 2.22: Tabla de operaciones para el sistema de números complejos.

Para implementar este plan, asumamos que tenemos dos procedimientos, `put` y `get`, para manipular la tabla de operaciones y tipos:

- $\hbox{\tt(put)\;\langle$*op*$\kern0.1em\rangle\;\langle$*tipo*$ \kern0.08em\rangle\;\langle$*item*$\kern0.08em\rangle\hbox{\tt)}$ instala el ⟨*item*$\kern0.08em\rangle$ en la tabla, indexado por ⟨*op*$ \kern0.1em\rangle$ y ⟨*tipo*$\kern0.08em\rangle$.

- $\hbox{\tt(get)\;\langle$*op*$\kern0.1em\rangle\;\langle$*tipo*$ \kern0.08em\rangle\hbox{\tt)}$ busca la entrada ⟨*op*$\kern0.1em \rangle$, ⟨*tipo*$\kern0.08em\rangle$ en la tabla y devuelve el item encontrado allí. Si no se encuentra ningún item, `get` devuelve falso.

Por ahora, podemos asumir que `put` y `get` están incluidos en nuestro lenguaje. En el Capítulo 3 (Section 3.3.3) veremos cómo implementar estas y otras operaciones para manipular tablas.

Así es como se puede usar la programación dirigida por datos en el sistema de números complejos. Ben, quien desarrolló la representación rectangular, implementa su código tal como lo hizo originalmente. Define una colección de procedimientos, o un *paquete*, e interconecta estos con el resto del sistema agregando entradas a la tabla que le dicen al sistema cómo operar sobre los números rectangulares. Esto se logra llamando al siguiente procedimiento:

(define (install-rectangular-package) ;; procedimientos internos (define

(real-part z) (car z)) (define (imag-part z) (cdr z)) (define (make-from-real-imag x y) (cons x y)) (define (magnitude z) (sqrt (+ (square (real-part z)) (square (imag-part z))))) (define (angle z) (atan (imag-part z) (real-part z))) (define (make-from-mag-ang r a) (cons (* r (cos a)) (* r (sin a))))

;; interfaz con el resto del sistema (define (tag x) (attach-tag 'rectangular x)) (put 'real-part '(rectangular) real-part) (put 'imag-part '(rectangular) imag-part) (put 'magnitude '(rectangular) magnitude) (put 'angle '(rectangular) angle) (put 'make-from-real-imag 'rectangular (lambda (x y) (tag (make-from-real-imag x y)))) (put 'make-from-mag-ang 'rectangular (lambda (r a) (tag (make-from-mag-ang r a)))) 'done)

Nótese que los procedimientos internos aquí son los mismos procedimientos de Section 2.4.1 que Ben escribió cuando trabajaba de forma aislada. No se necesitan cambios para interconectarlos con el resto del sistema. Además, dado que estas definiciones de procedimientos son internas al procedimiento de instalación, Ben no necesita preocuparse por conflictos de nombres con otros procedimientos fuera del paquete rectangular. Para interconectar estos con el resto del sistema, Ben instala su procedimiento `real/part` bajo el nombre de operación `real/part` y el tipo `(rectangular)`, y de manera similar para los otros selectores.[45] La interfaz también define los constructores que serán utilizados por el sistema externo.[46] Estos son idénticos a los constructores definidos internamente por Ben, excepto que adjuntan la etiqueta.

El paquete polar de Alyssa es análogo:

(define (install-polar-package) ;; procedimientos internos (define (magnitude z) (car z)) (define (angle z) (cdr z)) (define (make-from-mag-ang r a) (cons r a)) (define (real-part z) (* (magnitude z) (cos (angle z)))) (define (imag-part z) (* (magnitude z) (sin (angle z)))) (define (make-from-real-imag x y) (cons (sqrt (+ (square x) (square y))) (atan y x))) ;; interfaz con el resto del sistema (define (tag x) (attach-tag 'polar x)) (put 'real-part '(polar) real-part) (put 'imag-part '(polar) imag-part) (put 'magnitude '(polar) magnitude) (put 'angle '(polar) angle) (put 'make-from-real-imag 'polar (lambda (x y) (tag (make-from-real-imag x y)))) (put 'make-from-mag-ang 'polar (lambda (r a) (tag (make-from-mag-ang r a)))) 'done)

Aunque tanto Ben como Alyssa todavía usan sus procedimientos originales definidos con los mismos nombres que los del otro (p. ej., `real/part`), estas definiciones son ahora internas a diferentes procedimientos (ver Section 1.1.8), por lo que no hay conflicto de nombres.

Los selectores de aritmética compleja acceden a la tabla por medio de un

procedimiento general de "operación" llamado `apply/generic`, que aplica una operación genérica a algunos argumentos. `apply/generic` busca en la tabla bajo el nombre de la operación y los tipos de los argumentos y aplica el procedimiento resultante si hay uno presente:[47]

(define (apply-generic op . args) (let ((type-tags (map type-tag args))) (let ((proc (get op type-tags))) (if proc (apply proc (map contents args)) (error "No hay método para estos tipos: APPLY-GENERIC" (list op type-tags))))))

Usando `apply/generic`, podemos definir nuestros selectores genéricos de la siguiente manera:

(define (real-part z) (apply-generic 'real-part z)) (define (imag-part z) (apply-generic 'imag-part z)) (define (magnitude z) (apply-generic 'magnitude z)) (define (angle z) (apply-generic 'angle z))

Observe que estos no cambian en absoluto si se agrega una nueva representación al sistema.

También podemos extraer de la tabla los constructores que serán utilizados por los programas externos a los paquetes para crear números complejos a partir de partes reales e imaginarias y de magnitudes y ángulos. Como en Section 2.4.2, construimos números rectangulares siempre que tenemos partes reales e imaginarias, y números polares siempre que tenemos magnitudes y ángulos:

(define (make-from-real-imag x y) ((get 'make-from-real-imag 'rectangular) x y)) (define (make-from-mag-ang r a) ((get 'make-from-mag-ang 'polar) r a))

> **Ejercicio 2.73:** Section 2.3.2 describió un programa que realiza diferenciación simbólica:
>
> (define (deriv exp var) (cond ((number? exp) 0) ((variable? exp) (if (same-variable? exp var) 1 0)) ((sum? exp) (make-sum (deriv (addend exp) var) (deriv (augend exp) var))) ((product? exp) (make-sum (make-product (multiplier exp) (deriv (multiplicand exp) var)) (make-product (deriv (multiplier exp) var) (multiplicand exp)))) < *aquí se pueden agregar más reglas* > (else (error "tipo de expresión desconocido: DERIV" exp))))
>
> Podemos considerar este programa como realizando un despacho sobre el tipo de la expresión a diferenciar. En esta situación, la "etiqueta de tipo" del dato es el símbolo del operador algebraico (como `+`) y la operación que se realiza es `deriv`. Podemos transformar este programa a un estilo dirigido por datos reescribiendo el procedimiento de derivada básico

como

(define (deriv exp var) (cond ((number? exp) 0) ((variable? exp) (if (same-variable? exp var) 1 0)) (else ((get 'deriv (operator exp)) (operands exp) var)))) (define (operator exp) (car exp)) (define (operands exp) (cdr exp))

1. Explica lo que se hizo arriba. ¿Por qué no podemos asimilar los predicados `number?` y `variable?` en el despacho dirigido por datos?

2. Escribe los procedimientos para las derivadas de sumas y productos, y el código auxiliar requerido para instalarlos en la tabla utilizada por el programa anterior.

3. Elige cualquier regla de diferenciación adicional que te guste, como la de los exponentes (Ejercicio 2.56), e instálala en este sistema dirigido por datos.

4. En este simple manipulador algebraico, el tipo de una expresión es el operador algebraico que la une. Supongamos, sin embargo, que indexamos los procedimientos de la manera opuesta, de modo que la línea de despacho en `deriv` se viera así

   ((get (operator exp) 'deriv) (operands exp) var)

   ¿Qué cambios correspondientes al sistema de derivadas se requieren?

Ejercicio 2.74: Insaciable Enterprises, Inc., es una empresa conglomerada altamente descentralizada que consta de un gran número de divisiones independientes ubicadas en todo el mundo. Las instalaciones informáticas de la empresa acaban de ser interconectadas por medio de un ingenioso esquema de interfaz de red que hace que toda la red parezca a cualquier usuario ser una sola computadora. La presidenta de Insaciable, en su primer intento de explotar la capacidad de la red para extraer información administrativa de los archivos de las divisiones, se consterna al descubrir que, aunque todos los archivos de las divisiones han sido implementados como estructuras de datos en Scheme, la estructura de datos particular utilizada varía de división a división. Se convoca apresuradamente una reunión de gerentes de división para buscar una estrategia para integrar los archivos que satisfaga las necesidades de la sede central preservando al mismo tiempo la autonomía existente de las divisiones.

Muestre cómo se puede implementar tal estrategia con programación dirigida por datos. Como ejemplo, suponga que los registros de personal

de cada división consisten en un solo archivo, que contiene un conjunto de registros con clave en los nombres de los empleados. La estructura del conjunto varía de división a división. Además, el registro de cada empleado es en sí mismo un conjunto (estructurado de manera diferente de división a división) que contiene información con claves bajo identificadores como `address` y `salary`. En particular:

1. Implemente para la sede central un procedimiento `get/record` que recupere el registro de un empleado específico de un archivo de personal específico. El procedimiento debería ser aplicable al archivo de cualquier división. Explique cómo deben estructurarse los archivos de las divisiones individuales. En particular, ¿qué información de tipo debe proporcionarse?

2. Implemente para la sede central un procedimiento `get/salary` que devuelva la información salarial del registro de un empleado dado de cualquier archivo de personal de la división. ¿Cómo debe estructurarse el registro para que esta operación funcione?

3. Implemente para la sede central un procedimiento `find/employee/record`. Este debería buscar en todos los archivos de las divisiones el registro de un empleado dado y devolver el registro. Suponga que este procedimiento toma como argumentos el nombre de un empleado y una lista de todos los archivos de las divisiones.

4. Cuando Insaciable se hace cargo de una nueva empresa, ¿qué cambios deben realizarse para incorporar la nueva información de personal en el sistema central?

Paso de mensajes

La idea clave de la programación dirigida por datos es manejar las operaciones genéricas en los programas tratando explícitamente con tablas de operaciones y tipos, como la tabla en la Figura 2.22. El estilo de programación que usamos en Section 2.4.2 organizaba el despacho requerido según el tipo haciendo que cada operación se encargara de su propio despacho. En efecto, esto descompone la tabla de operaciones y tipos en filas, donde cada procedimiento de operación genérica representa una fila de la tabla.

Una estrategia de implementación alternativa es descomponer la tabla en columnas y, en lugar de usar "operaciones inteligentes" que despachan según los tipos de datos, trabajar con "objetos de datos inteligentes" que despachan según los nombres de las operaciones. Podemos hacer esto organizando las

cosas de modo que un objeto de datos, como un número rectangular, se represente como un procedimiento que toma como entrada el nombre de la operación requerida y realiza la operación indicada. En tal disciplina, `make/from/real/imag` podría escribirse como

(define (make-from-real-imag x y) (define (dispatch op) (cond ((eq? op 'real-part) x) ((eq? op 'imag-part) y) ((eq? op 'magnitude) (sqrt (+ (square x) (square y)))) ((eq? op 'angle) (atan y x)) (else (error "Op desconocido: MAKE-FROM-REAL-IMAG" op)))) dispatch)

El procedimiento `apply/generic` correspondiente, que aplica una operación genérica a un argumento, ahora simplemente le pasa el nombre de la operación al objeto de datos y deja que el objeto haga el trabajo:[48]

(define (apply-generic op arg) (arg op))

Nótese que el valor devuelto por `make/from/real/imag` es un procedimiento —el procedimiento interno `dispatch`. Este es el procedimiento que se invoca cuando `apply/generic` solicita que se realice una operación.

Este estilo de programación se llama *paso de mensajes*. El nombre proviene de la imagen de que un objeto de datos es una entidad que recibe el nombre de la operación solicitada como un "mensaje". Ya hemos visto un ejemplo de paso de mensajes en Section 2.1.3, donde vimos cómo `cons`, `car` y `cdr` podían definirse sin objetos de datos, solo con procedimientos. Aquí vemos que el paso de mensajes no es un truco matemático sino una técnica útil para organizar sistemas con operaciones genéricas. En el resto de este capítulo continuaremos usando la programación dirigida por datos, en lugar del paso de mensajes, para discutir las operaciones aritméticas genéricas. En el Capítulo 3 volveremos al paso de mensajes, y veremos que puede ser una herramienta poderosa para estructurar programas de simulación.

Ejercicio 2.75: Implemente el constructor `make/from/mag/ang` en estilo de pa-so de men-sa-jes. Este procedimiento debería ser análogo al procedimiento `make/from/real/imag` dado anteriormente.

Ejercicio 2.76: A medida que un sistema grande con operaciones genéricas evoluciona, pueden ser necesarios nuevos tipos de objetos de datos o nuevas operaciones. Para cada una de las tres estrategias—operaciones genéricas con despacho explícito, estilo dirigido por datos y estilo de paso de mensajes—describa los cambios que deben hacerse en un sistema para agregar nuevos tipos o nuevas operaciones. ¿Qué organización sería más apropiada para un sistema en el que se deben agregar nuevos tipos con frecuencia? ¿Cuál sería más apropiada para un sistema en el que se deben agregar nuevas operaciones con frecuencia?

Sistemas con Operaciones Genéricas

En la sección anterior, vimos cómo diseñar sistemas en los que los objetos de datos pueden ser representados en más de una forma. La idea clave es vincular el código que especifica las operaciones de datos a las diversas representaciones por medio de procedimientos de interfaz genéricos. Ahora veremos cómo usar esta misma idea no solo para definir operaciones que son genéricas sobre diferentes representaciones sino también para definir operaciones que son genéricas sobre diferentes tipos de argumentos. Ya hemos visto varios paquetes diferentes de operaciones aritméticas: la aritmética primitiva (`+`, `-`, `*`, `/`) integrada en nuestro lenguaje, la aritmética de números racionales (`add/rat`, `sub/rat`, `mul/rat`, `div/rat`) de Section 2.1.1, y la aritmética de números complejos que implementamos en Section 2.4.3. Ahora usaremos técnicas dirigidas por datos para construir un paquete de operaciones aritméticas que incorpora todos los paquetes aritméticos que ya hemos construido.

Figura 2.23: Sistema de aritmética genérica.

La Figura 2.23 muestra la estructura del sistema que construiremos. Nótese las barreras de abstracción. Desde la perspectiva de alguien que usa "números", hay un único procedimiento `add` que opera sobre cualquier número que se le suministre. `add` es parte de una interfaz genérica que permite que los paquetes de arit-mé-ti-ca or-di-na-ria, arit-mé-ti-ca ra-cio-nal y arit-mé-ti-ca com-ple-ja sean accedidos de manera uniforme por programas que usan números. Cualquier paquete aritmético individual (como el paquete complejo) puede a su vez ser accedido a través de procedimientos genéricos (como `add/complex`) que combinan paquetes diseñados para diferentes representaciones (como rectangular y polar). Además, la estructura del sistema es aditiva, de modo que se pueden diseñar los paquetes aritméticos individuales por separado y combinarlos para producir un sistema de aritmética genérica.

Operaciones Aritméticas Genéricas

La tarea de diseñar operaciones aritméticas genéricas es análoga a la de diseñar las operaciones genéricas de números complejos. Nos gustaría, por ejemplo, tener un procedimiento de adición genérico `add` que actúe como la adición primitiva ordinaria `+` en números ordinarios, como `add/rat` en números racionales, y como `add/complex` en números complejos. Podemos implementar `add`, y las otras operaciones aritméticas genéricas, siguiendo la

misma estrategia que usamos en Section 2.4.3 para implementar los selectores genéricos para números complejos. Adjuntaremos una etiqueta de tipo a cada tipo de número y haremos que el procedimiento genérico despache a un paquete apropiado según el tipo de dato de sus argumentos.

Los procedimientos aritméticos genéricos se definen de la siguiente manera:

(define (add x y) (apply-generic 'add x y)) (define (sub x y) (apply-generic 'sub x y)) (define (mul x y) (apply-generic 'mul x y)) (define (div x y) (apply-generic 'div x y))

Comenzamos instalando un paquete para manejar números *ordinarios*, es decir, los números primitivos de nuestro lenguaje. Los etiquetaremos con el símbolo `scheme/number`. Las operaciones aritméticas en este paquete son los procedimientos aritméticos primitivos (por lo que no hay necesidad de definir procedimientos adicionales para manejar los números sin etiquetar). Como estas operaciones toman cada una dos argumentos, se instalan en la tabla con la clave de la lista `(scheme/number scheme/number)`:

(define (install-scheme-number-package) (define (tag x) (attach-tag 'scheme-number x)) (put 'add '(scheme-number scheme-number) (lambda (x y) (tag (+ x y)))) (put 'sub '(scheme-number scheme-number) (lambda (x y) (tag (- x y)))) (put 'mul '(scheme-number scheme-number) (lambda (x y) (tag (* x y)))) (put 'div '(scheme-number scheme-number) (lambda (x y) (tag (/ x y)))) (put 'make 'scheme-number (lambda (x) (tag x))) 'done)

Los usuarios del paquete Scheme-number crearán números ordinarios (etiquetados) por medio del procedimiento:

(define (make-scheme-number n) ((get 'make 'scheme-number) n))

Ahora que el marco del sistema de aritmética genérica está en su lugar, podemos incluir fácilmente nuevos tipos de números. Aquí hay un paquete que realiza aritmética racional. Nótese que, como beneficio de la aditividad, podemos usar sin modificación el código de números racionales de Section 2.1.1 como los procedimientos internos en el paquete:

(define (install-rational-package) ;; procedimientos internos (define (numer x) (car x)) (define (denom x) (cdr x)) (define (make-rat n d) (let ((g (gcd n d))) (cons (/ n g) (/ d g)))) (define (add-rat x y) (make-rat (+ (* (numer x) (denom y)) (* (numer y) (denom x))) (* (denom x) (denom y)))) (define (sub-rat x y) (make-rat (- (* (numer x) (denom y)) (* (numer y) (denom x))) (* (denom x) (denom y)))) (define (mul-rat x y) (make-rat (* (numer x) (numer y)) (* (denom x) (denom y)))) (define (div-rat x y) (make-rat (* (numer x) (denom y)) (* (denom x) (numer y)))) ;; interfaz con el resto del sistema

(define (tag x) (attach-tag 'rational x)) (put 'add '(rational rational) (lambda (x y) (tag (add-rat x y)))) (put 'sub '(rational rational) (lambda (x y) (tag (sub-rat x y)))) (put 'mul '(rational rational) (lambda (x y) (tag (mul-rat x y)))) (put 'div '(rational rational) (lambda (x y) (tag (div-rat x y)))) (put 'make 'rational (lambda (n d) (tag (make-rat n d)))) 'done) (define (make-rational n d) ((get 'make 'rational) n d))

Podemos instalar un paquete similar para manejar números complejos, usando la etiqueta `complex`. Al crear el paquete, extraemos de la tabla las operaciones `make/from/real/imag` y `make/from/mag/ang` que fueron definidas por los paquetes rectangular y polar. La aditividad nos permite usar, como operaciones internas, los mismos procedimientos `add/complex`, `sub/complex`, `mul/complex` y `div/complex` de Section 2.4.1.

(define (install-complex-package) ;; procedimientos importados de los paquetes rectangular y polar (define (make-from-real-imag x y) ((get 'make-from-real-imag 'rectangular) x y)) (define (make-from-mag-ang r a) ((get 'make-from-mag-ang 'polar) r a)) ;; procedimientos internos (define (add-complex z1 z2) (make-from-real-imag (+ (real-part z1) (real-part z2)) (+ (imag-part z1) (imag-part z2)))) (define (sub-complex z1 z2) (make-from-real-imag (- (real-part z1) (real-part z2)) (- (imag-part z1) (imag-part z2)))) (define (mul-complex z1 z2) (make-from-mag-ang (* (magnitude z1) (magnitude z2)) (+ (angle z1) (angle z2)))) (define (div-complex z1 z2) (make-from-mag-ang (/ (magnitude z1) (magnitude z2)) (- (angle z1) (angle z2)))) ;; interfaz con el resto del sistema (define (tag z) (attach-tag 'complex z)) (put 'add '(complex complex) (lambda (z1 z2) (tag (add-complex z1 z2)))) (put 'sub '(complex complex) (lambda (z1 z2) (tag (sub-complex z1 z2)))) (put 'mul '(complex complex) (lambda (z1 z2) (tag (mul-complex z1 z2)))) (put 'div '(complex complex) (lambda (z1 z2) (tag (div-complex z1 z2)))) (put 'make-from-real-imag 'complex (lambda (x y) (tag (make-from-real-imag x y)))) (put 'make-from-mag-ang 'complex (lambda (r a) (tag (make-from-mag-ang r a)))) 'done)

Los programas fuera del paquete de números complejos pueden construir números complejos ya sea a partir de partes reales e imaginarias o a partir de magnitudes y ángulos. Nótese cómo los procedimientos subyacentes, originalmente definidos en los paquetes rectangular y polar, se exportan al paquete complejo, y de ahí se exportan al mundo exterior.

(define (make-complex-from-real-imag x y) ((get 'make-from-real-imag 'complex) x y)) (define (make-complex-from-mag-ang r a) ((get 'make-from-mag-ang 'complex) r a))

Lo que tenemos aquí es un sistema de etiquetas de dos niveles. Un número

complejo típico, como 3 + 4*i* en forma rectangular, se representaría como se muestra en la Figura 2.24. La etiqueta externa (`complex`) se utiliza para dirigir el número al paquete complejo. Una vez dentro del paquete complejo, la siguiente etiqueta (`rectangular`) se utiliza para dirigir el número al paquete rectangular. En un sistema grande y complicado podría haber muchos niveles, cada uno interconectado con el siguiente por medio de operaciones genéricas. A medida que un objeto de datos se pasa "hacia abajo", la etiqueta externa que se utiliza para dirigirlo al paquete apropiado se elimina (aplicando `contents`) y el siguiente nivel de etiqueta (si lo hay) se vuelve visible para ser utilizado para un despacho posterior.

Figura 2.24: Representación de 3 + 4*i* en forma rectangular.

En los paquetes anteriores, usamos `add/rat`, `add/complex` y los otros procedimientos aritméticos exactamente como se escribieron originalmente. Sin embargo, una vez que estas definiciones son internas a diferentes procedimientos de instalación, ya no necesitan nombres que sean distintos entre sí: podríamos simplemente nombrarlos `add`, `sub`, `mul` y `div` en ambos paquetes.

Ejercicio 2.77: Louis Reasoner intenta evaluar la expresión `(magnitude z)` donde `z` es el objeto que se muestra en la Figura 2.24. Para su sorpresa, en lugar de la respuesta 5 obtiene un mensaje de error de `apply/generic`, diciendo que no hay método para la operación `magnitude` en los tipos `(complex)`. Le muestra esta interacción a Alyssa P. Hacker, quien dice: "El problema es que los selectores de números complejos nunca se definieron para números `complex`, solo para números `polar` y `rectangular`. Todo lo que tienes que hacer para que esto funcione es agregar lo siguiente al paquete `complex`:"

(put 'real-part '(complex) real-part) (put 'imag-part '(complex) imag-part) (put 'magnitude '(complex) magnitude) (put 'angle '(complex) angle)

Describe en detalle por qué esto funciona. Como ejemplo, traza todos los procedimientos llamados al evaluar la expresión `(magnitude z)` donde `z` es el objeto que se muestra en la Figura 2.24. En particular, ¿cuántas veces se invoca `apply/generic`? ¿A qué procedimiento se despacha en cada caso?

Ejercicio 2.78: Los procedimientos internos en el paquete `scheme/number` son esencialmente nada más que llamadas a los procedimientos primitivos `+`, `-`, etc. No fue posible usar los primitivos del lenguaje directamente porque nuestro sistema de etiquetas de tipo requiere que

cada objeto de datos tenga un tipo adjunto. De hecho, sin embargo, todas las implementaciones de Lisp tienen un sistema de tipos, que utilizan internamente. Predicados primitivos como `symbol?` y `number?` determinan si los objetos de datos tienen tipos particulares. Modifica las definiciones de `type/tag`, `contents` y `attach/tag` de Section 2.4.2 para que nuestro sistema genérico aproveche el sistema de tipos interno de Scheme. Es decir, el sistema debería funcionar como antes, excepto que los números ordinarios deberían estar representados simplemente como números de Scheme en lugar de como pares cuyo `car` es el símbolo `scheme/number`.

Ejercicio 2.79: Define un predicado de igualdad genérico `equ?` que pruebe la igualdad de dos números, e instálalo en el paquete de aritmética genérica. Esta operación debería funcionar para números ordinarios, números racionales y números complejos.

Ejercicio 2.80: Define un predicado genérico `=zero?` que pruebe si su argumento es cero, e instálalo en el paquete de aritmética genérica. Esta operación debería funcionar para números ordinarios, números racionales y números complejos.

Combinando Datos de Diferentes Tipos

Hemos visto cómo definir un sistema aritmético unificado que abarca números ordinarios, números complejos, números racionales y cualquier otro tipo de número que decidamos inventar, pero hemos ignorado un tema importante. Las operaciones que hemos definido hasta ahora tratan los diferentes tipos de datos como completamente independientes. Así, hay paquetes separados para sumar, digamos, dos números ordinarios, o dos números complejos. Lo que aún no hemos considerado es el hecho de que es significativo definir operaciones que cruzan los límites de tipo, como la adición de un número complejo a un número ordinario. Hemos hecho grandes esfuerzos para introducir barreras entre las partes de nuestros programas para que puedan desarrollarse y entenderse por separado. Nos gustaría introducir las operaciones entre tipos de una manera cuidadosamente controlada, para que podamos soportarlas sin violar seriamente nuestros límites de módulo.

Una forma de manejar las operaciones entre tipos es diseñar un procedimiento diferente para cada combinación posible de tipos para los que la operación es válida. Por ejemplo, podríamos extender el paquete de números complejos para que proporcione un procedimiento para sumar números complejos a números ordinarios e instalarlo en la tabla usando la

etiqueta `(complex scheme/number)`:[49]

```
;; para ser incluido en el paquete complejo
(define (add-complex-to-schemenum z x)
  (make-from-real-imag (+ (real-part z) x) (imag-part z)))
(put 'add '(complex scheme-number)
     (lambda (z x) (tag (add-complex-to-schemenum z x))))
```

Esta técnica funciona, pero es engorrosa. Con tal sistema, el costo de introducir un nuevo tipo no es solo la construcción del paquete de procedimientos para ese tipo, sino también la construcción e instalación de los procedimientos que implementan las operaciones entre tipos. Esto puede ser fácilmente mucho más código del necesario para definir las operaciones en el tipo mismo. El método también socava nuestra capacidad para combinar paquetes separados de forma aditiva, o al menos para limitar la medida en que los implementadores de los paquetes individuales necesitan tener en cuenta otros paquetes. Por ejemplo, en el ejemplo anterior, parece razonable que el manejo de operaciones mixtas sobre números complejos y números ordinarios sea responsabilidad del paquete de números complejos. Sin embargo, combinar números racionales y números complejos podría ser hecho por el paquete complejo, por el paquete racional, o por algún tercer paquete que use operaciones extraídas de estos dos paquetes. Formular políticas coherentes sobre la división de responsabilidades entre paquetes puede ser una tarea abrumadora en el diseño de sistemas con muchos paquetes y muchas operaciones entre tipos.

Coerción

En la situación general de operaciones completamente no relacionadas que actúan sobre tipos completamente no relacionados, implementar operaciones explícitas entre tipos, por engorroso que pueda ser, es lo mejor que se puede esperar. Afortunadamente, generalmente podemos hacerlo mejor aprovechando la estructura adicional que pueda estar latente en nuestro sistema de tipos. A menudo, los diferentes tipos de datos no son completamente independientes, y puede haber formas en las que los objetos de un tipo puedan ser vistos como siendo de otro tipo. Este proceso se llama *coerción*. Por ejemplo, si se nos pide combinar aritméticamente un número ordinario con un número complejo, podemos ver el número ordinario como un número complejo cuya parte imaginaria es cero. Esto transforma el problema al de combinar dos números complejos, lo que puede ser manejado de la manera ordinaria por el paquete de aritmética compleja.

En general, podemos implementar esta idea diseñando procedimientos de coerción que transformen un objeto de un tipo en un objeto equivalente de otro tipo. Aquí hay un procedimiento de coerción típico, que transforma un

número ordinario dado en un número complejo con esa parte real y parte imaginaria cero:

(define (scheme-number->complex n) (make-complex-from-real-imag (contents n) 0))

Instalamos estos procedimientos de coerción en una tabla de coerción especial, indexada bajo los nombres de los dos tipos:

(put-coercion 'scheme-number 'complex scheme-number->complex)

(Asumimos que hay procedimientos `put/coercion` y `get/coercion` disponibles para manipular esta tabla.) Generalmente algunas de las ranuras en la tabla estarán vacías, porque generalmente no es posible coaccionar un objeto de datos arbitrario de cada tipo a todos los otros tipos. Por ejemplo, no hay forma de coaccionar un número complejo arbitrario a un número ordinario, por lo que no habrá un procedimiento general `complex/>scheme/number` incluido en la tabla.

Una vez que la tabla de coerción ha sido configurada, podemos manejar la coerción de una manera uniforme modificando el procedimiento `apply/generic` de Section 2.4.3. Cuando se nos pide aplicar una operación, primero verificamos si la operación está definida para los tipos de los argumentos, tal como antes. Si es así, despachamos al procedimiento encontrado en la tabla de operaciones y tipos. De lo contrario, intentamos la coerción. Por simplicidad, consideramos solo el caso donde hay dos argumentos.[50] Verificamos la tabla de coerción para ver si los objetos del primer tipo pueden ser coaccionados al segundo tipo. Si es así, coaccionamos el primer argumento e intentamos la operación de nuevo. Si los objetos del primer tipo no pueden en general ser coaccionados al segundo tipo, intentamos la coerción al revés para ver si hay una forma de coaccionar el segundo argumento al tipo del primer argumento. Finalmente, si no hay forma conocida de coaccionar ningún tipo al otro tipo, nos damos por vencidos. Aquí está el procedimiento:

(define (apply-generic op . args) (let ((type-tags (map type-tag args))) (let ((proc (get op type-tags))) (if proc (apply proc (map contents args)) (if (= (length args) 2) (let ((type1 (car type-tags)) (type2 (cadr type-tags)) (a1 (car args)) (a2 (cadr args))) (let ((t1->t2 (get-coercion type1 type2)) (t2->t1 (get-coercion type2 type1))) (cond (t1->t2 (apply-generic op (t1->t2 a1) a2)) (t2->t1 (apply-generic op a1 (t2->t1 a2))) (else (error "No hay método para estos tipos" (list op type-tags)))))) (error "No hay método para estos tipos" (list op type-tags)))))))

Este esquema de coerción tiene muchas ventajas sobre el método de definir

operaciones explícitas entre tipos, como se describió anteriormente. Aunque todavía necesitamos escribir procedimientos de coerción para relacionar los tipos (posiblemente n^2 procedimientos para un sistema con n tipos), solo necesitamos escribir un procedimiento por cada par de tipos en lugar de un procedimiento diferente para cada colección de tipos y cada operación genérica.[51] En lo que contamos aquí es en el hecho de que la transformación apropiada entre tipos depende solo de los tipos mismos, no de la operación a aplicar.

Por otro lado, puede haber aplicaciones para las cuales nuestro esquema de coerción no es lo suficientemente general. Incluso cuando ninguno de los objetos a combinar puede ser convertido al tipo del otro, todavía puede ser posible realizar la operación convirtiendo ambos objetos a un tercer tipo. Para lidiar con tal complejidad y aún preservar la modularidad en nuestros programas, generalmente es necesario construir sistemas que aprovechen aún más estructura en las relaciones entre tipos, como discutimos a continuación.

Jerarquías de tipos

El esquema de coerción presentado anteriormente se basaba en la existencia de relaciones naturales entre pares de tipos. A menudo hay una estructura más "global" en cómo los diferentes tipos se relacionan entre sí. Por ejemplo, supongamos que estamos construyendo un sistema aritmético genérico para manejar enteros, números racionales, números reales y números complejos. En tal sistema, es bastante natural considerar un entero como un tipo especial de número racional, que a su vez es un tipo especial de número real, que a su vez es un tipo especial de número complejo. Lo que realmente tenemos es una llamada *jerarquía de tipos*, en la que, por ejemplo, los enteros son un *subtipo* de los números racionales (es decir, cualquier operación que se pueda aplicar a un número racional se puede aplicar automáticamente a un entero). A la inversa, decimos que los números racionales forman un *supertipo* de los enteros. La jerarquía particular que tenemos aquí es de un tipo muy simple, en el que cada tipo tiene como máximo un supertipo y como máximo un subtipo. Tal estructura, llamada una *torre*, se ilustra en la Figura 2.25.

Figura 2.25: Una torre de tipos.

Si tenemos una estructura de torre, entonces podemos simplificar enormemente el problema de agregar un nuevo tipo a la jerarquía, ya que solo necesitamos especificar cómo se incrusta el nuevo tipo en el siguiente supertipo por encima de él y cómo es el supertipo del tipo por debajo de él. Por ejemplo, si queremos agregar un entero a un número complejo, no

necesitamos definir explícitamente un procedimiento de coerción especial `integer/>complex`. En cambio, definimos cómo un entero puede ser transformado en un número racional, cómo un número racional se transforma en un número real, y cómo un número real se transforma en un número complejo. Luego permitimos que el sistema transforme el entero en un número complejo a través de estos pasos y luego sume los dos números complejos.

Podemos rediseñar nuestro procedimiento `apply/generic` de la siguiente manera: Para cada tipo, necesitamos proporcionar un procedimiento `raise`, que "eleva" los objetos de ese tipo un nivel en la torre. Luego, cuando se requiere que el sistema opere sobre objetos de diferentes tipos, puede elevar sucesivamente los tipos inferiores hasta que todos los objetos estén al mismo nivel en la torre. (El Ejercicio 2.83 y el Ejercicio 2.84 se refieren a los detalles de implementar tal estrategia.)

Otra ventaja de una torre es que podemos implementar fácilmente la noción de que cada tipo "hereda" todas las operaciones definidas en un supertipo. Por ejemplo, si no proporcionamos un procedimiento especial para encontrar la parte real de un entero, deberíamos esperar sin embargo que `real/part` esté definida para los enteros en virtud del hecho de que los enteros son un subtipo de los números complejos. En una torre, podemos arreglar que esto suceda de una manera uniforme modificando `apply/generic`. Si la operación requerida no está directamente definida para el tipo del objeto dado, elevamos el objeto a su supertipo e intentamos de nuevo. Así, subimos por la torre, transformando nuestro argumento a medida que avanzamos, hasta que encontramos un nivel en el que se puede realizar la operación deseada o llegamos a la cima (en cuyo caso nos damos por vencidos).

Otra ventaja más de una torre sobre una jerarquía más general es que nos da una forma simple de "bajar" un objeto de datos a la representación más simple. Por ejemplo, si sumamos $2 + 3i$ a $4 - 3i$, sería bueno obtener la respuesta como el entero 6 en lugar de como el número complejo $6 + 0i$. El Ejercicio 2.85 discute una forma de implementar tal operación de descenso. (El truco es que necesitamos una forma general de distinguir aquellos objetos que pueden ser bajados, como $6 + 0i$, de aquellos que no, como $6 + 2i$.)

Figura 2.26: Relaciones entre tipos de figuras geométricas.

Insuficiencias de las jerarquías

Si los tipos de datos en nuestro sistema pueden organizarse naturalmente en

una torre, esto simplifica enormemente los problemas de tratar con operaciones genéricas sobre diferentes tipos, como hemos visto. Desafortunadamente, este no suele ser el caso. La Figura 2.26 ilustra una disposición más compleja de tipos mixtos, esta mostrando relaciones entre diferentes tipos de figuras geométricas. Vemos que, en general, un tipo puede tener más de un subtipo. Los triángulos y cuadriláteros, por ejemplo, son ambos subtipos de polígonos. Además, un tipo puede tener más de un supertipo. Por ejemplo, un triángulo rectángulo isósceles puede ser considerado como un triángulo isósceles o como un triángulo rectángulo. Este problema de múltiples supertipos es particularmente espinoso, ya que significa que no hay una forma única de "elevar" un tipo en la jerarquía. Encontrar el supertipo "correcto" en el cual aplicar una operación a un objeto puede implicar una búsqueda considerable a través de toda la red de tipos por parte de un procedimiento como `apply/generic`. Dado que generalmente hay múltiples subtipos para un tipo, existe un problema similar al coaccionar un valor "hacia abajo" en la jerarquía de tipos. Lidiar con un gran número de tipos interrelacionados mientras se preserva la modularidad en el diseño de grandes sistemas es muy difícil, y es un área de mucha investigación actual.[52]

Ejercicio 2.81: Louis Reasoner ha notado que `apply/generic` puede intentar coaccionar los argumentos al tipo del otro incluso si ya tienen el mismo tipo. Por lo tanto, razona, necesitamos poner procedimientos en la tabla de coerción para *coaccionar* argumentos de cada tipo a su propio tipo. Por ejemplo, además de la coerción `scheme/number/>complex` mostrada arriba, él haría:

(define (scheme-number-> scheme-number n) n) (define (complex-> complex z) z) (put-coercion 'scheme-number 'scheme-number scheme-number-> scheme-number) (put-coercion 'complex 'complex complex-> complex)

1. Con los procedimientos de coerción de Louis instalados, ¿qué sucede si se llama a `apply/generic` con dos argumentos de tipo `scheme/number` o dos argumentos de tipo `complex` para una operación que no se encuentra en la tabla para esos tipos? Por ejemplo, asumamos que hemos definido una operación de exponenciación genérica:

   (define (exp x y) (apply-generic 'exp x y))

   y hemos puesto un procedimiento para la exponenciación en el paquete Scheme-number pero no en ningún otro paquete:

   ;; lo siguiente se agregó al paquete Scheme-number (put 'exp

'(scheme-number scheme-number) (lambda (x y) (tag (expt x y)))) ; usando el primitivo `expt`

¿Qué sucede si llamamos a `exp` con dos números complejos como argumentos?

2. ¿Está en lo correcto Louis al pensar que había que hacer algo con la coerción con argumentos del mismo tipo, o `apply/generic` funciona correctamente tal como está?

3. Modifica `apply/generic` para que no intente la coerción si los dos argumentos tienen el mismo tipo.

Ejercicio 2.82: Muestra cómo generalizar `apply/generic` para manejar la coerción en el caso general de múltiples argumentos. Una estrategia es intentar coaccionar todos los argumentos al tipo del primer argumento, luego al tipo del segundo argumento, y así sucesivamente. Da un ejemplo de una situación donde esta estrategia (y de igual manera la versión de dos argumentos dada arriba) no es suficientemente general. (Pista: Considera el caso donde hay algunas operaciones de tipo mixto adecuadas presentes en la tabla que no se intentarán.)

Ejercicio 2.83: Supongamos que estás diseñando un sistema aritmético genérico para tratar con la torre de tipos que se muestra en la Figura 2.25: entero, racional, real, complejo. Para cada tipo (excepto complejo), diseña un procedimiento que eleve los objetos de ese tipo un nivel en la torre. Muestra cómo instalar una operación genérica `raise` que funcionará para cada tipo (excepto complejo).

Ejercicio 2.84: Usando la operación `raise` del Ejercicio 2.83, modifica el procedimiento `apply/generic` para que coaccione sus argumentos para que tengan el mismo tipo por el método de elevación sucesiva, como se discute en esta sección. Necesitarás idear una forma de probar cuál de dos tipos es más alto en la torre. Haz esto de una manera que sea "compatible" con el resto del sistema y que no cause problemas al agregar nuevos niveles a la torre.

Ejercicio 2.85: Esta sección mencionó un método para "simplificar" un objeto de datos bajándolo en la torre de tipos tanto como sea posible. Diseña un procedimiento `drop` que logre esto para la torre descrita en el Ejercicio 2.83. La clave es decidir, de alguna manera general, si un objeto puede ser bajado. Por ejemplo, el número complejo $1.5 + 0i$ puede bajarse hasta `real`, el número complejo $1 + 0i$ puede bajarse hasta `integer`, y el número complejo $2 + 3i$ no puede bajarse en absoluto. Aquí hay un plan para determinar si un objeto puede ser bajado:

Comienza por definir una operación genérica `project` que "empuja" un objeto hacia abajo en la torre. Por ejemplo, proyectar un número complejo implicaría descartar la parte imaginaria. Entonces un número puede ser bajado si, cuando lo `proyectamos` y `elevamos` el resultado de vuelta al tipo con el que empezamos, terminamos con algo igual a lo que empezamos. Muestra cómo implementar esta idea en detalle, escribiendo un procedimiento `drop` que baje un objeto tanto como sea posible. Necesitarás diseñar las diversas operaciones de proyección[53] e instalar `project` como una operación genérica en el sistema. También necesitarás hacer uso de un predicado de igualdad genérico, como el descrito en el Ejercicio 2.79. Finalmente, usa `drop` para reescribir `apply/generic` del Ejercicio 2.84 para que "simplifique" sus respuestas.

Ejercicio 2.86: Supongamos que queremos manejar números complejos cuyas partes reales, partes imaginarias, magnitudes y ángulos pueden ser números ordinarios, números racionales u otros números que podríamos desear agregar al sistema. Describe e implementa los cambios en el sistema necesarios para acomodar esto. Tendrás que definir operaciones como `sine` y `cosine` que sean genéricas sobre números ordinarios y números racionales.

Ejemplo: Álgebra Simbólica

La manipulación de expresiones algebraicas simbólicas es un proceso complejo que ilustra muchos de los problemas más difíciles que ocurren en el diseño de sistemas a gran escala. Una expresión algebraica, en general, puede verse como una estructura jerárquica, un árbol de operadores aplicados a operandos. Podemos construir expresiones algebraicas comenzando con un conjunto de objetos primitivos, como constantes y variables, y combinándolos por medio de operadores algebraicos, como la suma y la multiplicación. Como en otros lenguajes, formamos abstracciones que nos permiten referirnos a objetos compuestos en términos simples. Las abstracciones típicas en el álgebra simbólica son ideas como combinación lineal, polinomio, función racional o función trigonométrica. Podemos considerar estas como "tipos" compuestos, que a menudo son útiles para dirigir el procesamiento de expresiones. Por ejemplo, podríamos describir la expresión

$$x^2 \sin(y^2 + 1) + x \cos 2y + \cos(y^3 - 2y^2)$$

como un polinomio en x con coeficientes que son funciones trigonométricas de polinomios en y cuyos coeficientes son enteros.

No intentaremos desarrollar un sistema completo de manipulación algebraica

aquí. Tales sistemas son programas extremadamente complejos, que encarnan un profundo conocimiento algebraico y algoritmos elegantes. Lo que haremos es examinar una parte simple pero importante de la manipulación algebraica: la aritmética de polinomios. Ilustraremos los tipos de decisiones que enfrenta el diseñador de tal sistema, y cómo aplicar las ideas de datos abstractos y operaciones genéricas para ayudar a organizar este esfuerzo.

Aritmética sobre polinomios

Nuestra primera tarea al diseñar un sistema para realizar aritmética sobre polinomios es decidir qué es exactamente un polinomio. Los polinomios se definen normalmente en relación con ciertas variables (las *indeterminadas* del polinomio). Por simplicidad, nos restringiremos a polinomios con una sola indeterminada (*polinomios univariados*).[54] Definiremos un polinomio como una suma de términos, cada uno de los cuales es un coeficiente, una potencia de la indeterminada, o un producto de un coeficiente y una potencia de la indeterminada. Un coeficiente se define como una expresión algebraica que no depende de la indeterminada del polinomio. Por ejemplo,

$$5x^2 + 3x + 7$$

es un polinomio simple en x, y

$$(y^2 + 1)x^3 + (2y)x + 1$$

es un polinomio en x cuyos coeficientes son polinomios en y.

Ya estamos bordeando algunos temas espinosos. ¿Es el primero de estos polinomios lo mismo que el polinomio $5y^2 + 3y + 7$, o no? Una respuesta razonable podría ser "sí, si estamos considerando un polinomio puramente como una función matemática, pero no, si estamos considerando que un polinomio es una forma sintáctica". El segundo polinomio es algebraicamente equivalente a un polinomio en y cuyos coeficientes son polinomios en x. ¿Debería nuestro sistema reconocer esto, o no? Además, hay otras formas de representar un polinomio—por ejemplo, como un producto de factores, o (para un polinomio univariado) como el conjunto de raíces, o como una lista de los valores del polinomio en un conjunto especificado de puntos.[55] Podemos sortear estas preguntas decidiendo que en nuestro sistema de manipulación algebraica un "polinomio" será una forma sintáctica particular, no su significado matemático subyacente.

Ahora debemos considerar cómo realizar la aritmética en polinomios. En este sistema simple, consideraremos solo la suma y la multiplicación. Además, insistiremos en que dos polinomios a combinar deben tener la misma

indeterminada.

Abordaremos el diseño de nuestro sistema siguiendo la disciplina familiar de la abstracción de datos. Representaremos los polinomios usando una estructura de datos llamada *poli*, que consiste en una variable y una colección de términos. Asumiremos que tenemos los selectores `variable` y `term/list` que extraen esas partes de un poli y un constructor `make/poly` que ensambla un poli a partir de una variable y una lista de términos dados. Una variable será solo un símbolo, por lo que podemos usar el procedimiento `same/variable?` de Section 2.3.2 para comparar variables. Los siguientes procedimientos definen la suma y la multiplicación de polis:

(define (add-poly p1 p2) (if (same-variable? (variable p1) (variable p2)) (make-poly (variable p1) (add-terms (term-list p1) (term-list p2))) (error "Polis no en la misma var: ADD-POLY" (list p1 p2)))) (define (mul-poly p1 p2) (if (same-variable? (variable p1) (variable p2)) (make-poly (variable p1) (mul-terms (term-list p1) (term-list p2))) (error "Polis no en la misma var: MUL-POLY" (list p1 p2))))

Para incorporar polinomios en nuestro sistema de aritmética genérica, necesitamos proporcionarles etiquetas de tipo. Usaremos la etiqueta `polynomial`, e instalaremos operaciones apropiadas en polinomios etiquetados en la tabla de operaciones. Incrustaremos todo nuestro código en un procedimiento de instalación para el paquete de polinomios, similar a los de Section 2.5.1:

(define (install-polynomial-package) ;; procedimientos internos ;; representación de poli (define (make-poly variable term-list) (cons variable term-list)) (define (variable p) (car p)) (define (term-list p) (cdr p)) < *procedimientos* same-variable? *y* variable? *de la sección 2.3.2* > ;; representación de términos y listas de términos < *procedimientos* adjoin-term … coeff *del texto a continuación* > (define (add-poly p1 p2) …) < *procedimientos usados por* add-poly > (define (mul-poly p1 p2) …) < *procedimientos usados por* mul-poly > ;; interfaz con el resto del sistema (define (tag p) (attach-tag 'polynomial p)) (put 'add '(polynomial polynomial) (lambda (p1 p2) (tag (add-poly p1 p2)))) (put 'mul '(polynomial polynomial) (lambda (p1 p2) (tag (mul-poly p1 p2)))) (put 'make 'polynomial (lambda (var terms) (tag (make-poly var terms)))) 'done)

La suma de polinomios se realiza término a término. Los términos del mismo orden (es decir, con la misma potencia de la indeterminada) deben combinarse. Esto se hace formando un nuevo término del mismo orden cuyo coeficiente es la suma de los coeficientes de los sumandos. Los términos en un sumando para los cuales no hay términos del mismo orden en el otro

sumando se acumulan simplemente en el polinomio suma que se está construyendo.

Para manipular listas de términos, asumiremos que tenemos un constructor `the/empty/termlist` que devuelve una lista de términos vacía y un constructor `adjoin/term` que adjunta un nuevo término a una lista de términos. También asumiremos que tenemos un predicado `empty/termlist?` que nos dice si una lista de términos dada está vacía, un selector `first/term` que extrae el término de orden más alto de una lista de términos, y un selector `rest/terms` que devuelve todos menos el término de orden más alto. Para manipular términos, supondremos que tenemos un constructor `make/term` que construye un término con un orden y coeficiente dados, y los selectores `order` y `coeff` que devuelven, respectivamente, el orden y el coeficiente del término. Estas operaciones nos permiten considerar tanto los términos como las listas de términos como abstracciones de datos, cuyas representaciones concretas podemos preocuparnos por separado.

Aquí está el procedimiento que construye la lista de términos para la suma de dos polinomios:[56]

(define (add-terms L1 L2) (cond ((empty-termlist? L1) L2) ((empty-termlist? L2) L1) (else (let ((t1 (first-term L1)) (t2 (first-term L2))) (cond ((> (order t1) (order t2)) (adjoin-term t1 (add-terms (rest-terms L1) L2))) ((< (order t1) (order t2)) (adjoin-term t2 (add-terms L1 (rest-terms L2)))) (else (adjoin-term (make-term (order t1) (add (coeff t1) (coeff t2))) (add-terms (rest-terms L1) (rest-terms L2)))))))))

El punto más importante a tener en cuenta aquí es que usamos el procedimiento de suma genérico `add` para sumar los coeficientes de los términos que se están combinando. Esto tiene consecuencias poderosas, como veremos a continuación.

Para multiplicar dos listas de términos, multiplicamos cada término de la primera lista por todos los términos de la otra lista, usando repetidamente `mul/term/by/all/terms`, que multiplica un término dado por todos los términos en una lista de términos dada. Las listas de términos resultantes (una por cada término de la primera lista) se acumulan en una suma. Multiplicar dos términos forma un término cuyo orden es la suma de los órdenes de los factores y cuyo coeficiente es el producto de los coeficientes de los factores:

(define (mul-terms L1 L2) (if (empty-termlist? L1) (the-empty-termlist) (add-terms (mul-term-by-all-terms (first-term L1) L2) (mul-terms (rest-terms L1) L2)))) (define (mul-term-by-all-terms t1 L) (if (empty-termlist? L) (the-empty-

termlist) (let ((t2 (first-term L))) (adjoin-term (make-term (+ (order t1) (order t2)) (mul (coeff t1) (coeff t2))) (mul-term-by-all-terms t1 (rest-terms L))))))

Esto es realmente todo lo que hay en la suma y multiplicación de polinomios. Nótese que, dado que operamos sobre términos usando los procedimientos genéricos `add` y `mul`, nuestro paquete de polinomios es automáticamente capaz de manejar cualquier tipo de coeficiente que sea conocido por el paquete de aritmética genérica. Si incluimos un mecanismo de coerción como uno de los discutidos en Section 2.5.2, entonces también somos automáticamente capaces de manejar operaciones sobre polinomios de diferentes tipos de coeficientes, como

$$[3x^2 + (2 + 3i)x + 7] \cdot \! \left[x^4 + {2\over3} x^2 + (5 + 3i) \right]\!.$$

Debido a que instalamos los procedimientos de suma y multiplicación de polinomios `add/poly` y `mul/poly` en el sistema de aritmética genérica como las operaciones `add` y `mul` para el tipo `polynomial`, nuestro sistema también es automáticamente capaz de manejar operaciones polinómicas como

$$[(y + 1)x^2 + (y^2 + 1)x + (y - 1)] \cdot [(y - 2)x + (y^3 + 7)].$$

La razón es que cuando el sistema intenta combinar coeficientes, despachará a través de `add` y `mul`. Dado que los coeficientes son ellos mismos polinomios (en y), estos se combinarán usando `add/poly` y `mul/poly`. El resultado es una especie de "recursión dirigida por datos" en la que, por ejemplo, una llamada a `mul/poly` resultará en llamadas recursivas a `mul/poly` para multiplicar los coeficientes. Si los coeficientes de los coeficientes fueran ellos mismos polinomios (como podría usarse para representar polinomios en tres variables), la dirección de datos aseguraría que el sistema seguiría a través de otro nivel de llamadas recursivas, y así sucesivamente a través de tantos niveles como la estructura de los datos dicte.[57]

Representando listas de términos

Finalmente, debemos enfrentar la tarea de implementar una buena representación para las listas de términos. Una lista de términos es, en efecto, un conjunto de coeficientes con clave por el orden del término. Por lo tanto, cualquiera de los métodos para representar conjuntos, como se discutió en Section 2.3.3, puede aplicarse a esta tarea. Por otro lado, nuestros procedimientos `add/terms` y `mul/terms` siempre acceden a las listas de términos secuencialmente desde el orden más alto al más bajo. Por lo tanto, usaremos algún tipo de representación de lista ordenada.

¿Cómo deberíamos estructurar la lista que representa una lista de términos? Una consideración es la "densidad" de los polinomios que pretendemos manipular. Se dice que un polinomio es *denso* si tiene coeficientes no nulos en términos de la mayoría de los órdenes. Si tiene muchos términos cero, se dice que es *disperso*. Por ejemplo,

$$A: \quad x^5 + 2x^4 + 3x^2 - 2x - 5$$

es un polinomio denso, mientras que

$$B: \quad x^{100} + 2x^2 + 1$$

es disperso.

Las listas de términos de los polinomios densos se representan de manera más eficiente como listas de los coeficientes. Por ejemplo, *A* arriba estaría bien representado como `(1 2 0 3 -2 -5)`. El orden de un término en esta representación es la longitud de la sublista que comienza con el coeficiente de ese término, decrementada en 1.[58] Esta sería una representación terrible para un polinomio disperso como *B*: habría una lista gigante de ceros puntuada por unos pocos términos no nulos solitarios. Una representación más razonable de la lista de términos de un polinomio disperso es como una lista de los términos no nulos, donde cada término es una lista que contiene el orden del término y el coeficiente para ese orden. En tal esquema, el polinomio *B* se representa eficientemente como `((100 1) (2 2) (0 1))`. Como la mayoría de las manipulaciones de polinomios se realizan en polinomios dispersos, usaremos este método. Asumiremos que las listas de términos se representan como listas de términos, ordenadas desde el término de orden más alto al de orden más bajo. Una vez que hemos tomado esta decisión, implementar los selectores y constructores para términos y listas de términos es sencillo:[59]

(define (adjoin-term term term-list) (if (=zero? (coeff term)) term-list (cons term term-list))) (define (the-empty-termlist) '()) (define (first-term term-list) (car term-list)) (define (rest-terms term-list) (cdr term-list)) (define (empty-termlist? term-list) (null? term-list)) (define (make-term order coeff) (list order coeff)) (define (order term) (car term)) (define (coeff term) (cadr term))

donde `=zero?` es como se define en el Ejercicio 2.80. (Ver también el Ejercicio 2.87 a continuación.)

Los usuarios del paquete de polinomios crearán polinomios (etiquetados) por medio del procedimiento:

```
(define (make-polynomial var terms) ((get 'make 'polynomial) var terms))
```

Ejercicio 2.87: Instala `=zero?` para polinomios en el paquete de aritmética genérica. Esto permitirá que `adjoin/term` funcione para polinomios con coeficientes que son ellos mismos polinomios.

Ejercicio 2.88: Extiende el sistema de polinomios para incluir la resta de polinomios. (Pista: puede que te resulte útil definir una operación de negación genérica.)

Ejercicio 2.89: Define procedimientos que implementen la representación de lista de términos descrita anteriormente como apropiada para polinomios densos.

Ejercicio 2.90: Supongamos que queremos tener un sistema de polinomios que sea eficiente tanto para polinomios dispersos como densos. Una forma de hacer esto es permitir ambos tipos de representaciones de listas de términos en nuestro sistema. La situación es análoga al ejemplo de números complejos de Section 2.4, donde permitimos tanto representaciones rectangulares como polares. Para hacer esto, debemos distinguir diferentes tipos de listas de términos y hacer que las operaciones en listas de términos sean genéricas. Rediseña el sistema de polinomios para implementar esta generalización. Este es un esfuerzo mayor, no un cambio local.

Ejercicio 2.91: Un polinomio univariado puede dividirse por otro para producir un cociente polinómico y un resto polinómico. Por ejemplo,

$$${x^5 - 1 \over x^2 - 1} = x^3 + x, \hbox{ resto } x - 1.$$$

La división se puede realizar mediante división larga. Es decir, divide el término de mayor orden del dividendo por el término de mayor orden del divisor. El resultado es el primer término del cociente. A continuación, multiplica el resultado por el divisor, réstalo del dividendo y produce el resto de la respuesta dividiendo recursivamente la diferencia por el divisor. Detente cuando el orden del divisor exceda el orden del dividendo y declara el dividendo como el resto. Además, si el dividendo alguna vez se vuelve cero, devuelve cero como cociente y resto.

Podemos diseñar un procedimiento `div/poly` sobre el modelo de `add/poly` y `mul/poly`. El procedimiento comprueba si los dos polis tienen la misma variable. Si es así, `div/poly` quita la variable y pasa el problema a `div/terms`, que realiza la operación de división en las listas de términos. `div/poly` finalmente vuelve a adjuntar la variable al resultado proporcionado por `div/terms`. Es conveniente diseñar `div/terms` para

calcular tanto el cociente como el resto de una división. `div/terms` puede tomar dos listas de términos como argumentos y devolver una lista de la lista de términos del cociente y la lista de términos del resto.

Completa la siguiente definición de `div/terms` rellenando las expresiones faltantes. Usa esto para implementar `div/poly`, que toma dos polis como argumentos y devuelve una lista de los polis cociente y resto.

(define (div-terms L1 L2) (if (empty-termlist? L1) (list (the-empty-termlist) (the-empty-termlist)) (let ((t1 (first-term L1)) (t2 (first-term L2))) (if (> (order t2) (order t1)) (list (the-empty-termlist) L1) (let ((new-c (div (coeff t1) (coeff t2))) (new-o (- (order t1) (order t2)))) (let ((rest-of-result ⟨ *calcula el resto del resultado recursivamente* ⟩)) ⟨ *forma el resultado completo* ⟩))))))

Jerarquías de tipos en álgebra simbólica

Nuestro sistema de polinomios ilustra cómo los objetos de un tipo (polinomios) pueden de hecho ser objetos complejos que tienen objetos de muchos tipos diferentes como partes. Esto no plantea ninguna dificultad real en la definición de operaciones genéricas. Solo necesitamos instalar operaciones genéricas apropiadas para realizar las manipulaciones necesarias de las partes de los tipos compuestos. De hecho, vimos que los polinomios forman una especie de "abstracción de datos recursiva", en el sentido de que partes de un polinomio pueden ser ellas mismas polinomios. Nuestras operaciones genéricas y nuestro estilo de programación dirigida por datos pueden manejar esta complicación sin mucho problema.

Por otro lado, el álgebra de polinomios es un sistema para el cual los tipos de datos no pueden organizarse naturalmente en una torre. Por ejemplo, es posible tener polinomios en *x* cuyos coeficientes son polinomios en *y*. También es posible tener polinomios en *y* cuyos coeficientes son polinomios en *x*. Ninguno de estos tipos está "por encima" del otro de ninguna manera natural, sin embargo, a menudo es necesario sumar elementos de cada conjunto. Hay varias formas de hacer esto. Una posibilidad es convertir un polinomio al tipo del otro expandiendo y reorganizando los términos para que ambos polinomios tengan la misma variable principal. Se puede imponer una estructura similar a una torre a esto ordenando las variables y, por lo tanto, siempre convirtiendo cualquier polinomio a una "forma canónica" con la variable de mayor prioridad dominante y las variables de menor prioridad enterradas en los coeficientes. Esta estrategia funciona bastante bien, excepto que la conversión puede expandir un polinomio innecesariamente, haciéndolo difícil de leer y quizás menos eficiente para trabajar con él. La estrategia de la torre ciertamente no es natural para este dominio o para

cualquier dominio donde el usuario puede inventar nuevos tipos dinámicamente usando tipos antiguos en varias formas de combinación, como funciones trigonométricas, series de potencias e integrales.

No debería sorprender que controlar la coerción sea un problema serio en el diseño de sistemas de manipulación algebraica a gran escala. Gran parte de la complejidad de tales sistemas se refiere a las relaciones entre diversos tipos. De hecho, es justo decir que aún no entendemos completamente la coerción. De hecho, aún no entendemos completamente el concepto de tipo de dato. Sin embargo, lo que sabemos nos proporciona poderosos principios de estructuración y modularidad para apoyar el diseño de grandes sistemas.

> **Ejercicio 2.92:** Imponiendo un orden a las variables, extiende el paquete de polinomios para que la suma y la multiplicación de polinomios funcione para polinomios en diferentes variables. (¡Esto no es fácil!)

Ejercicio extendido: Funciones racionales

Podemos extender nuestro sistema de aritmética genérica para incluir *funciones racionales*. Estas son "fracciones" cuyo numerador y denominador son polinomios, como

$$${x + 1 \over x^3 - 1}\,.$$

El sistema debería poder sumar, restar, multiplicar y dividir funciones racionales, y realizar cálculos como

$$${x + 1 \over x^3 - 1} + {x \over x^2 - 1} = {x^3 + 2x^2 + 3x + 1 \over x^4 + x^3 - x - 1}\,.$$

(Aquí la suma ha sido simplificada eliminando factores comunes. La "multiplicación cruzada" ordinaria habría producido un polinomio de cuarto grado sobre un polinomio de quinto grado.)

Si modificamos nuestro paquete de aritmética racional para que use operaciones genéricas, entonces hará lo que queremos, excepto por el problema de reducir fracciones a su mínima expresión.

> **Ejercicio 2.93:** Modifica el paquete de aritmética racional para usar operaciones genéricas, pero cambia `make/rat` para que no intente reducir las fracciones a su mínima expresión. Prueba tu sistema llamando a `make/rational` en dos polinomios para producir una función racional:
>
> (define p1 (make-polynomial 'x '((2 1) (0 1)))) (define p2 (make-polynomial 'x '((3 1) (0 1)))) (define rf (make-rational p2 p1))

Ahora suma `rf` a sí mismo, usando `add`. Observarás que este procedimiento de suma no reduce las fracciones a su mínima expresión.

Podemos reducir fracciones polinómicas a su mínima expresión usando la misma idea que usamos con los enteros: modificando `make/rat` para dividir tanto el numerador como el denominador por su máximo común divisor. La noción de "máximo común divisor" tiene sentido para los polinomios. De hecho, podemos calcular el MCD de dos polinomios usando esencialmente el mismo Algoritmo de Euclides que funciona para los enteros.[60] La versión para enteros es

(define (gcd a b) (if (= b 0) a (gcd b (remainder a b))))

Usando esto, podríamos hacer la modificación obvia para definir una operación MCD que funcione en listas de términos:

(define (gcd-terms a b) (if (empty-termlist? b) a (gcd-terms b (remainder-terms a b))))

donde `remainder/terms` selecciona el componente del resto de la lista devuelta por la operación de división de listas de términos `div/terms` que se implementó en el Ejercicio 2.91.

Ejercicio 2.94: Usando `div/terms`, implementa el procedimiento `remainder/terms` y úsalo para definir `gcd/terms` como anteriormente. Ahora escribe un procedimiento `gcd/poly` que calcule el MCD polinómico de dos polis. (El procedimiento debería señalar un error si los dos polis no están en la misma variable.) Instala en el sistema una operación genérica `greatest/common/divisor` que se reduzca a `gcd/poly` para polinomios y al `gcd` ordinario para números ordinarios. Como prueba, intenta

(define p1 (make-polynomial 'x '((4 1) (3 -1) (2 -2) (1 2)))) (define p2 (make-polynomial 'x '((3 1) (1 -1)))) (greatest-common-divisor p1 p2)

y comprueba tu resultado a mano.

Ejercicio 2.95: Define P_1, P_2, y P_3 como los polinomios

$$\begin{array}{l@{{}:}l} P_1 & \quad x^2 - 2x + 1, \\ P_2 & \quad 11x^2 + 7, \\ P_3 & \quad 13x + 5. \end{array}$$

Ahora define Q_1 como el producto de P_1 y P_2 y Q_2 como el producto de P_1 y P_3, y usa `greatest/common/divisor` (Ejercicio 2.94) para calcular el MCD de Q_1 y Q_2. Nótese que la respuesta no es la misma que P_1. Este

ejemplo introduce operaciones no enteras en el cálculo, causando dificultades con el algoritmo MCD.[61] Para entender lo que está sucediendo, intenta trazar `gcd/terms` mientras calculas el MCD o intenta realizar la división a mano.

Podemos resolver el problema exhibido en el Ejercicio 2.95 si usamos la siguiente modificación del algoritmo MCD (que realmente solo funciona en el caso de polinomios con coeficientes enteros). Antes de realizar cualquier división de polinomios en el cálculo del MCD, multiplicamos el dividendo por un factor constante entero, elegido para garantizar que no surgirán fracciones durante el proceso de división. Nuestra respuesta diferirá así del MCD real por un factor constante entero, pero esto no importa en el caso de reducir funciones racionales a su mínima expresión; el MCD se usará para dividir tanto el numerador como el denominador, por lo que el factor constante entero se cancelará.

Más precisamente, si P y Q son polinomios, sea O_1 el orden de P (es decir, el orden del término más grande de P) y sea O_2 el orden de Q. Sea c el coeficiente principal de Q. Entonces se puede demostrar que, si multiplicamos P por el *factor de enterización* $c^{1 + O_1 - O_2}$, el polinomio resultante puede ser dividido por Q usando el algoritmo `div/terms` sin introducir ninguna fracción. La operación de multiplicar el dividendo por esta constante y luego dividir se llama a veces la *pseudodivisión* de P por Q. El resto de la división se llama el *pseudoresto*.

Ejercicio 2.96:

1. Implementa el procedimiento `pseudoremainder/terms`, que es igual que `remainder/terms` excepto que multiplica el dividendo por el factor de enterización descrito anteriormente antes de llamar a `div/terms`. Modifica `gcd/terms` para usar `pseudoremainder/terms`, y verifica que `greatest/common/divisor` ahora produce una respuesta con coeficientes enteros en el ejemplo del Ejercicio 2.95.

2. El MCD ahora tiene coeficientes enteros, pero son más grandes que los de P_1. Modifica `gcd/terms` para que elimine los factores comunes de los coeficientes de la respuesta dividiendo todos los coeficientes por su máximo común divisor (entero).

Así, aquí está cómo reducir una función racional a su mínima expresión:

- Calcula el MCD del numerador y el denominador, usando la versión de `gcd/terms` del Ejercicio 2.96.

- Cuando obtengas el MCD, multiplica tanto el numerador como el

denominador por el mismo factor de enterización antes de dividir por el MCD, para que la división por el MCD no introduzca ningún coeficiente no entero. Como factor puedes usar el coeficiente principal del MCD elevado a la potencia $1 + O_1 - O_2$, donde O_2 es el orden del MCD y O_1 es el máximo de los órdenes del numerador y el denominador. Esto asegurará que dividir el numerador y el denominador por el MCD no introducirá ninguna fracción.

- El resultado de esta operación será un numerador y denominador con coeficientes enteros. Los coeficientes normalmente serán muy grandes debido a todos los factores de enterización, por lo que el último paso es eliminar los factores redundantes calculando el máximo común divisor (entero) de todos los coeficientes del numerador y el denominador y dividiendo por este factor.

Ejercicio 2.97:

1. Implementa este algoritmo como un procedimiento `reduce/terms` que toma dos listas de términos `n` y `d` como argumentos y devuelve una lista `nn`, `dd`, que son `n` y `d` reducidos a su mínima expresión mediante el algoritmo dado anteriormente. También escribe un procedimiento `reduce/poly`, análogo a `add/poly`, que comprueba si los dos polis tienen la misma variable. Si es así, `reduce/poly` quita la variable y pasa el problema a `reduce/terms`, luego vuelve a adjuntar la variable a las dos listas de términos proporcionadas por `reduce/terms`.

2. Define un procedimiento análogo a `reduce/terms` que haga lo que el `make/rat` original hacía para los enteros:

   (define (reduce-integers n d) (let ((g (gcd n d))) (list (/ n g) (/ d g))))

   y define `reduce` como una operación genérica que llama a `apply/generic` para despachar ya sea a `reduce/poly` (para argumentos `polynomial`) o a `reduce/integers` (para argumentos `scheme/number`). Ahora puedes hacer fácilmente que el paquete de aritmética racional reduzca las fracciones a su mínima expresión haciendo que `make/rat` llame a `reduce` antes de combinar el numerador y el denominador dados para formar un número racional. El sistema ahora maneja expresiones racionales ya sea en enteros o en polinomios. Para probar tu programa, intenta el ejemplo al principio de este ejercicio extendido:

   (define p1 (make-polynomial 'x '((1 1) (0 1)))) (define p2 (make-polynomial 'x '((3 1) (0 -1)))) (define p3 (make-polynomial 'x '((1

1)))) (define p4 (make-polynomial 'x '((2 1) (0 -1)))) (define rf1 (make-rational p1 p2)) (define rf2 (make-rational p3 p4)) (add rf1 rf2)

Mira si obtienes la respuesta correcta, correctamente reducida a su mínima expresión.

El cálculo del MCD está en el corazón de cualquier sistema que realice operaciones sobre funciones racionales. El algoritmo usado anteriormente, aunque matemáticamente sencillo, es extremadamente lento. La lentitud se debe en parte al gran número de operaciones de división y en parte al enorme tamaño de los coeficientes intermedios generados por las pseudodivisiones. Una de las áreas activas en el desarrollo de sistemas de manipulación algebraica es el diseño de mejores algoritmos para calcular MCDs de polinomios.[62]

1. La capacidad de manipular procedimientos directamente proporciona un incremento análogo en la potencia expresiva de un lenguaje de programación. Por ejemplo, en Section 1.3.1 introducimos el procedimiento `sum`, que toma un procedimiento `term` como argumento y calcula la suma de los valores de `term` sobre cierto intervalo especificado. A los fines de definir `sum`, resulta crucial que seamos capaces de expresarnos sobre un procedimiento como `term`, como una entidad en sí misma, sin prestar atención a cómo `term` se expresa en términos de operaciones más primitivas. Por supuesto, si no disponemos de la noción de "un procedimiento", dudo que podamos incluso pensar en la posibilidad de definir una operación como `sum`. Además, en lo que respecta a la sumatoria, resultan irrelevantes los detalles de la contrucción de `term` a partir de las operaciones más primitivas.

2. El nombre `cons` significa "construir". Los nombres `car` y `cdr` derivan de la implementación original de Lisp en el IBM 704. Esa máquina tenía un esquema de direccionamiento que permitía hacer referencia a las partes de "dirección" y "decremento" de una ubicación de memoria. `car` significa "Contenido de la parte de Dirección del Registro" y `cdr` (pronunciado "could-er") significa "Contenido de la parte de Decremento del Registro".

3. Otra forma de definir los selectores y el constructor es

(define make-rat cons) (define numer car) (define denom cdr)

La primera definición asocia el nombre `make/rat` con el valor de la expresión `cons`, que es el procedimiento primitivo que construye pares. Por lo tanto, `make/rat` y `cons` son nombres para el mismo constructor primitivo.

Definir selectores y constructores de esta manera es eficiente: en lugar de que `make/rat` *llame* a `cons`, `make/rat` *es* `cons`, por lo que solo se llama a un procedimiento, no a dos, cuando se llama a `make/rat`. Por otro lado, hacer esto anula las ayudas de depuración que rastrean las llamadas a procedimientos o ponen puntos de interrupción en las llamadas a procedimientos: es posible que desee ver cómo se llama a `make/rat`, pero ciertamente no quiere ver cada llamada a `cons`.

Hemos optado por no usar este estilo de definición en este libro.

4. `display` es la primitiva de Scheme para imprimir datos. La primitiva de Scheme `newline` inicia una nueva línea para imprimir. Ninguno de estos procedimientos devuelve un valor útil, por lo que en los usos de `print/rat` a continuación, solo mostramos lo que `print/rat` imprime, no lo que el intérprete imprime como el valor devuelto por `print/rat`.

5. Sorprendentemente, esta idea es muy difícil de formular rigurosamente. Hay dos enfoques para dar tal formulación. Uno, iniciado por C. A. R. Hoare (1972), se conoce como el método de *modelos abstractos*. Formaliza la especificación de "procedimientos más condiciones" como se describe en el ejemplo de los números racionales anterior. Tenga en cuenta que la condición sobre la representación de los números racionales se estableció en términos de hechos sobre los enteros (igualdad y división). En general, los modelos abstractos definen nuevos tipos de objetos de datos en términos de tipos de objetos de datos previamente definidos. Las aserciones sobre los objetos de datos pueden, por lo tanto, verificarse reduciéndolas a aserciones sobre objetos de datos previamente definidos. Otro enfoque, introducido por Zilles en el MIT, por Goguen, Thatcher, Wagner y Wright en IBM (ver Thatcher et al. 1978), y por Guttag en Toronto (ver Guttag 1977), se llama *especificación algebraica*. Considera los "procedimientos" como elementos de un sistema algebraico abstracto cuyo comportamiento se especifica mediante axiomas que corresponden a nuestras "condiciones", y utiliza las técnicas del álgebra abstracta para verificar aserciones sobre objetos de datos. Ambos métodos se describen en el artículo de Liskov and Zilles (1975).

6. El uso de la palabra "clausura" aquí proviene del álgebra abstracta, donde se dice que un conjunto de elementos está cerrado bajo una operación si aplicar la operación a elementos del conjunto produce un elemento que vuelve a ser un elemento del conjunto. La comunidad de Lisp también (desafortunadamente) usa la palabra "clausura" para describir un concepto totalmente no relacionado: una clausura es una técnica de implementación para representar procedimientos con variables libres. No usamos la palabra "clausura" en este segundo sentido en este libro.

7. La noción de que un medio de combinación debe satisfacer la clausura es una idea sencilla. Desafortunadamente, los combinadores de datos proporcionados en muchos lenguajes de programación populares no satisfacen la clausura, o hacen que la clausura sea engorrosa de explotar. En Fortran o Basic, uno típicamente combina elementos de datos ensamblándolos en arreglos, pero no se pueden formar arreglos cuyos elementos sean a su vez arreglos. Pascal y C admiten estructuras cuyos elementos son estructuras. Sin embargo, esto requiere que el programador manipule punteros explícitamente y se adhiera a la restricción de que cada campo de una estructura solo puede contener elementos de una forma preespecificada. A diferencia de Lisp con sus pares, estos lenguajes no tienen un pegamento de propósito general incorporado que facilite la manipulación de datos compuestos de manera uniforme. Esta limitación subyace al comentario de Alan Perlis en su prólogo a este libro: "En Pascal, la plétora de estructuras de datos declarables induce una especialización dentro de las funciones que inhibe y penaliza la cooperación casual. Es mejor tener 100 funciones que operen sobre una estructura de datos que tener 10 funciones que operen sobre 10 estructuras de datos".

8. En este libro, usamos *lista* para referirnos a una cadena de pares terminada por el marcador de fin de lista. En contraste, el término *estructura de lista* se refiere a cualquier estructura de datos hecha de pares, no solo a listas.

9. Dado que las aplicaciones anidadas de `car` y `cdr` son engorrosas de escribir, los dialectos de Lisp proporcionan abreviaturas para ellas, por ejemplo,

(cadr ⟨ *arg* ⟩) = (car (cdr ⟨ *arg* ⟩))

Los nombres de todos esos procedimientos comienzan con `c` y terminan con `r`. Cada `a` entre ellos representa una operación `car` y cada `d` una operación `cdr`, que se aplicarán en el mismo orden en que aparecen en el nombre. Los nombres `car` y `cdr` persisten porque las combinaciones simples como `cadr` son pronunciables.

10. Es notable cuánta energía se ha disipado en la estandarización de los dialectos de Lisp en discusiones que son literalmente sobre nada: ¿Debería `nil` ser un nombre ordinario? ¿Debería el valor de `nil` ser un símbolo? ¿Debería ser una lista? ¿Debería ser un par? En Scheme, `nil` es un nombre ordinario, que usamos en esta sección como una variable cuyo valor es el marcador de fin de lista (al igual que `true` es una variable ordinaria que tiene un valor verdadero). Otros dialectos de Lisp, incluido Common Lisp, tratan `nil` como un símbolo especial. Los autores de este libro, que han soportado demasiadas peleas de estandarización de lenguajes, quisieran

evitar todo el asunto. Una vez que hayamos introducido la cita en Section 2.3, denotaremos la lista vacía como `'()` y prescindiremos por completo de la variable `nil`.

11. Para definir `f` y `g` usando `lambda` escribiríamos

(define f (lambda (x y . z) ⟨ *cuerpo* ⟩)) (define g (lambda w ⟨ *cuerpo* ⟩))

12. Scheme proporciona de forma estándar un procedimiento `map` que es más general que el descrito aquí. Este `map` más general toma un procedimiento de *n* argumentos, junto con *n* listas, y aplica el procedimiento a todos los primeros elementos de las listas, a todos los segundos elementos de las listas, y así sucesivamente, devolviendo una lista de los resultados. Por ejemplo:

(map + (list 1 2 3) (list 40 50 60) (list 700 800 900)) *(741 852 963)* (map (lambda (x y) (+ x (* 2 y))) (list 1 2 3) (list 4 5 6)) *(9 12 15)*

13. El orden de las dos primeras cláusulas en el `cond` es importante, ya que la lista vacía satisface `null?` y tampoco es un par.

14. Esto es, de hecho, precisamente el procedimiento `fringe` del Ejercicio 2.28. Aquí lo hemos renombrado para enfatizar que es parte de una familia de procedimientos generales de manipulación de secuencias.

15. Richard Waters (1979) desarrolló un programa que analiza automáticamente programas tradicionales de Fortran, viéndolos en términos de mapas, filtros y acumulaciones. Encontró que el 90 por ciento del código en el Paquete de Subrutinas Científicas de Fortran encaja perfectamente en este paradigma. Una de las razones del éxito de Lisp como lenguaje de programación es que las listas proporcionan un medio estándar para expresar colecciones ordenadas de modo que puedan ser manipuladas usando operaciones de orden superior. El lenguaje de programación APL debe gran parte de su poder y atractivo a una elección similar. En APL, todos los datos se representan como arreglos, y hay un conjunto universal y conveniente de operadores genéricos para todo tipo de operaciones de arreglos.

16. Según Knuth 1981, esta regla fue formulada por W. G. Horner a principios del siglo XIX, pero el método fue utilizado en realidad por Newton más de cien años antes. La regla de Horner evalúa el polinomio usando menos sumas y multiplicaciones que el método directo de calcular primero $a_n x^n$, luego sumar $a_{n-1} x^{n-1}$, y así sucesivamente. De hecho, es posible demostrar que cualquier algoritmo para evaluar polinomios arbitrarios debe usar al menos tantas sumas y multiplicaciones como la regla de Horner, y por lo tanto la regla de Horner es un algoritmo óptimo para la evaluación de polinomios. Esto fue demostrado (para el número de sumas) por A. M.

Ostrowski en un artículo de 1954 que esencialmente fundó el estudio moderno de los algoritmos óptimos. La afirmación análoga para las multiplicaciones fue demostrada por V. Y. Pan en 1966. El libro de Borodin and Munro (1975) proporciona una visión general de estos y otros resultados sobre algoritmos óptimos.

17. Esta definición utiliza la versión extendida de `map` descrita en la Footnote 12.

18. Este enfoque de mapeos anidados nos fue mostrado por David Turner, cuyos lenguajes KRC y Miranda proporcionan formalismos elegantes para tratar con estas construcciones. Los ejemplos en esta sección (ver también Ejercicio 2.42) están adaptados de Turner 1981. En Section 3.5.3, veremos cómo este enfoque se generaliza a secuencias infinitas.

19. Estamos representando un par aquí como una lista de dos elementos en lugar de como un par de Lisp. Por lo tanto, el "par" (i,j) se representa como `(list i j)`, no como `(cons i j)`.

20. El conjunto $S - x$ es el conjunto de todos los elementos de S, excluyendo x.

21. Los punto y coma en el código de Scheme se utilizan para introducir *comentarios*. Todo desde el punto y coma hasta el final de la línea es ignorado por el intérprete. En este libro no usamos muchos comentarios; intentamos que nuestros programas se autodocumenten usando nombres descriptivos.

22. El lenguaje de imágenes se basa en el lenguaje que Peter Henderson creó para construir imágenes como el grabado en madera "Límite Cuadrado" de M.C. Escher (ver Henderson 1982). El grabado incorpora un patrón escalado repetido, similar a las disposiciones dibujadas usando el procedimiento `square/limit` en esta sección.

23. William Barton Rogers (1804-1882) fue el fundador y primer presidente del MIT. Geólogo y talentoso profesor, enseñó en el William and Mary College y en la Universidad de Virginia. En 1859 se mudó a Boston, donde tuvo más tiempo para la investigación, trabajó en un plan para establecer un "instituto politécnico" y se desempeñó como el primer Inspector Estatal de Medidores de Gas de Massachusetts.

Cuando se estableció el MIT en 1861, Rogers fue elegido su primer presidente. Rogers defendió un ideal de "aprendizaje útil" que era diferente de la educación universitaria de la época, con su énfasis excesivo en los clásicos, que, como escribió, "se interponen en el camino de la instrucción y la disciplina más amplias, más elevadas y más prácticas de las ciencias

naturales y sociales". Esta educación también debía ser diferente de la estrecha educación de las escuelas de oficios. En palabras de Rogers:

> La distinción impuesta por el mundo entre el trabajador práctico y el científico es completamente fútil, y toda la experiencia de los tiempos modernos ha demostrado su absoluta inutilidad.

Rogers se desempeñó como presidente del MIT hasta 1870, cuando renunció por motivos de salud. En 1878, el segundo presidente del MIT, John Runkle, renunció bajo la presión de una crisis financiera provocada por el Pánico de 1873 y la tensión de luchar contra los intentos de Harvard de apoderarse del MIT. Rogers regresó para ocupar el cargo de presidente hasta 1881.

Rogers se desplomó y murió mientras se dirigía a la clase de graduados del MIT en las ceremonias de graduación de 1882. Runkle citó las últimas palabras de Rogers en un discurso conmemorativo pronunciado ese mismo año:

> "Mientras estoy aquí hoy y veo lo que es el Instituto, [...] recuerdo los comienzos de la ciencia. Recuerdo que hace ciento cincuenta años Stephen Hales publicó un folleto sobre el tema del gas de alumbrado, en el que afirmaba que sus investigaciones habían demostrado que 128 granos de carbón bituminoso..." "Carbón bituminoso", estas fueron sus últimas palabras en la tierra. Aquí se inclinó hacia adelante, como si consultara algunas notas en la mesa frente a él, luego, recuperando lentamente una posición erguida, levantó las manos y fue trasladado de la escena de sus trabajos y triunfos terrenales al "mañana de la muerte", donde se resuelven los misterios de la vida, y el espíritu desencarnado encuentra una satisfacción interminable en la contemplación de los nuevos y aún insondables misterios del futuro infinito.

En palabras de Francis A. Walker (tercer presidente del MIT):

> Toda su vida se había comportado con la mayor fidelidad y heroísmo, y murió como un caballero tan bueno seguramente habría deseado, en el arnés, en su puesto y en la parte y el acto mismo del deber público.

24. Equivalentemente, podríamos escribir

```
(define flipped-pairs (square-of-four identity flip-vert identity flip-vert))
```

25. `rotate180` rota un pintor 180 grados (ver Ejercicio 2.50). En lugar de `rotate180` podríamos decir `(compose flip/vert flip/horiz)`, usando el procedimiento `compose` del Ejercicio 1.42.

26. `frame/coord/map` usa las operaciones vectoriales descritas en el Ejercicio 2.46 a continuación, que asumimos que se han implementado usando alguna representación para vectores. Debido a la abstracción de datos, no importa cuál sea esta representación vectorial, siempre que las operaciones vectoriales se comporten correctamente.

27. `segments/>painter` usa la representación de segmentos de línea descrita en el Ejercicio 2.48 a continuación. También usa el procedimiento `for/each` descrito en el Ejercicio 2.23.

28. Por ejemplo, el pintor `rogers` de la Figura 2.11 se construyó a partir de una imagen en escala de grises. Para cada punto en un marco dado, el pintor `rogers` determina el punto en la imagen al que se mapea bajo el mapa de coordenadas del marco, y lo sombrea en consecuencia. Al permitir diferentes tipos de pintores, estamos aprovechando la idea de datos abstractos discutida en Section 2.1.3, donde argumentamos que una representación de números racionales podría ser cualquier cosa que satisfaga una condición apropiada. Aquí estamos usando el hecho de que un pintor puede implementarse de cualquier manera, siempre que dibuje algo en el marco designado. Section 2.1.3 también mostró cómo los pares podían implementarse como procedimientos. Los pintores son nuestro segundo ejemplo de una representación procedimental para datos.

29. `rotate90` es una rotación pura solo para marcos cuadrados, porque también estira y encoge la imagen para que quepa en el marco rotado.

30. Las imágenes en forma de diamante en la Figura 2.10 y la Figura 2.11 se crearon con `squash/inwards` aplicado a `wave` y `rogers`.

31. Section 3.3.4 describe uno de esos lenguajes.

32. Permitir la cita en un lenguaje causa estragos en la capacidad de razonar sobre el lenguaje en términos simples, porque destruye la noción de que los iguales pueden sustituirse por iguales. Por ejemplo, tres es uno más dos, pero la palabra "tres" no es la frase "uno más dos". La cita es poderosa porque nos da una forma de construir expresiones que manipulan otras expresiones (como veremos cuando escribamos un intérprete en el Capítulo 4). Pero permitir enunciados en un lenguaje que hablan de otros enunciados en ese lenguaje hace que sea muy difícil mantener cualquier principio coherente de lo que debería significar "los iguales pueden sustituirse por iguales". Por ejemplo, si sabemos que la estrella de la tarde es la estrella de la mañana, entonces del enunciado "la estrella de la tarde es Venus" podemos deducir "la estrella de la mañana es Venus". Sin embargo, dado que "Juan sabe que la estrella de la tarde es Venus" no podemos inferir que "Juan sabe que la

estrella de la mañana es Venus".

33. La comilla simple es diferente de la comilla doble que hemos estado usando para encerrar cadenas de caracteres que se imprimirán. Mientras que la comilla simple puede usarse para denotar listas o símbolos, la comilla doble se usa solo con cadenas de caracteres. En este libro, el único uso de las cadenas de caracteres es como elementos que se imprimirán.

34. Estrictamente, nuestro uso de la comilla viola la regla general de que todas las expresiones compuestas en nuestro lenguaje deben estar delimitadas por paréntesis y parecerse a listas. Podemos recuperar esta coherencia introduciendo una forma especial `quote`, que cumple el mismo propósito que la comilla. Así, escribiríamos `(quote a)` en lugar de `'a`, y escribiríamos `(quote (a b c))` en lugar de `'(a b c)`. Así es precisamente como funciona el intérprete. La comilla es solo una abreviatura de un solo carácter para envolver la siguiente expresión completa con `quote` para formar $ \hbox{\ttfamily(quote}\;\langle\kern0.06em\hbox{\ttfamily\slshape expression}\kern0.08em\rangle\hbox{\ttfamily)}$. Esto es importante porque mantiene el principio de que cualquier expresión vista por el intérprete puede manipularse como un objeto de datos. Por ejemplo, podríamos construir la expresión `(car '(a b c))`, que es lo mismo que `(car (quote (a b c)))`, evaluando `(list 'car (list 'quote '(a b c)))`.

35. Podemos considerar que dos símbolos son "el mismo" si consisten en los mismos caracteres en el mismo orden. Tal definición elude un problema profundo que aún no estamos listos para abordar: el significado de "igualdad" en un lenguaje de programación. Volveremos a esto en el Capítulo 3 (Section 3.1.3).

36. En la práctica, los programadores usan `equal?` para comparar listas que contienen números además de símbolos. Los números no se consideran símbolos. La cuestión de si dos números numéricamente iguales (como se prueba con `=`) también son `eq?` depende mucho de la implementación. Una mejor definición de `equal?` (como la que viene como primitiva en Scheme) también estipularía que si `a` y `b` son ambos números, entonces `a` y `b` son `equal?` si son numéricamente iguales.

37. Si queremos ser más formales, podemos especificar "consistente con las interpretaciones dadas anteriormente" para significar que las operaciones satisfacen una colección de reglas como estas:

• Para cualquier conjunto `S` y cualquier objeto `x`, `(element/of/set? x (adjoin/set x S))` es verdadero (informalmente: "Adjuntar un objeto a un conjunto produce un conjunto que contiene el objeto").

• Para cualquier conjunto `S` y `T` y cualquier objeto `x`, `(element/of/set? x (union/set S T))` es igual a `(or (element/of/set? x S) (element/of/set? x T))` (informalmente: "Los elementos de `(union S T)` son los elementos que están en `S` o en `T`").

• Para cualquier objeto `x`, `(element/of/set? x '())` es falso (informalmente: "Ningún objeto es un elemento del conjunto vacío").

38. Reducir a la mitad el tamaño del problema en cada paso es la característica distintiva del crecimiento logarítmico, como vimos con el algoritmo de exponenciación rápida de Section 1.2.4 y el método de búsqueda de bisección de Section 1.3.3.

39. Estamos representando conjuntos en términos de árboles, y árboles en términos de listas, en efecto, una abstracción de datos construida sobre otra abstracción de datos. Podemos considerar los procedimientos `entry`, `left/branch`, `right/branch` y `make/tree` como una forma de aislar la abstracción de un "árbol binario" de la forma particular en que podríamos desear representar tal árbol en términos de estructura de lista.

40. Ejemplos de tales estructuras incluyen los *árboles B* y los *árboles rojo-negro*. Existe una amplia literatura sobre estructuras de datos dedicada a este problema. Consulte Cormen et al. 1990.

41. Los Ejercicio 2.63 a Ejercicio 2.65 se deben a Paul Hilfinger.

42. Ver Hamming 1980 para una discusión de las propiedades matemáticas de los códigos de Huffman.

43. En sistemas computacionales reales, la forma rectangular es preferible a la forma polar la mayor parte del tiempo debido a los errores de redondeo en la conversión entre la forma rectangular y polar. Es por esto que el ejemplo de los números complejos es poco realista. Sin embargo, proporciona una ilustración clara del diseño de un sistema que utiliza operaciones genéricas y una buena introducción a los sistemas más sustanciales que se desarrollarán más adelante en este capítulo.

44. La función arcotangente a la que se hace referencia aquí, calculada por el procedimiento `atan` de Scheme, se define para tomar dos argumentos y y x y devolver el ángulo cuya tangente es y/x. Los signos de los argumentos determinan el cuadrante del ángulo.

45. Usamos la lista `(rectangular)` en lugar del símbolo `rectangular` para permitir la posibilidad de operaciones con múltiples argumentos, no todos del mismo tipo.

46. El tipo bajo el cual se instalan los constructores no necesita ser una lista porque un constructor siempre se usa para hacer un objeto de un tipo particular.

47. `apply/generic` usa la notación de cola punteada descrita en el Ejercicio 2.20, porque diferentes operaciones genéricas pueden tomar diferentes números de argumentos. En `apply/generic`, `op` tiene como valor el primer argumento de `apply/generic` y `args` tiene como valor una lista de los argumentos restantes.

`apply/generic` también usa el procedimiento primitivo `apply`, que toma dos argumentos, un procedimiento y una lista. `apply` aplica el procedimiento, usando los elementos de la lista como argumentos. Por ejemplo,

(apply + (list 1 2 3 4))

devuelve 10.

48. Una limitación de esta organización es que solo permite procedimientos genéricos de un argumento.

49. También tenemos que suministrar un procedimiento casi idéntico para manejar los tipos `(scheme/number complex)`.

50. Ver Ejercicio 2.82 para generalizaciones.

51. Si somos inteligentes, generalmente podemos arreglárnoslas con menos de n^2 procedimientos de coerción. Por ejemplo, si sabemos cómo convertir del tipo 1 al tipo 2 y del tipo 2 al tipo 3, entonces podemos usar este conocimiento para convertir del tipo 1 al tipo 3. Esto puede disminuir en gran medida el número de procedimientos de coerción que necesitamos suministrar explícitamente cuando agregamos un nuevo tipo al sistema. Si estamos dispuestos a incorporar la cantidad requerida de sofisticación en nuestro sistema, podemos hacer que busque en el "gráfico" de relaciones entre tipos y genere automáticamente aquellos procedimientos de coerción que pueden inferirse de los que se suministran explícitamente.

52. Esta declaración, que también aparece en la primera edición de este libro, es tan cierta ahora como lo era cuando la escribimos hace doce años. Desarrollar un marco útil y general para expresar las relaciones entre diferentes tipos de entidades (lo que los filósofos llaman "ontología") parece intratablemente difícil. La principal diferencia en-tre la confusión que existía hace diez años y la confusión que existe ahora es que ahora una variedad de teorías ontológicas inadecuadas han sido plasmadas en una plé-to-ra de lenguajes de programación correspondientemente inadecuados. Por ejemplo,

gran parte de la complejidad de los lenguajes de programación orientados a objetos —y las diferencias sutiles y confusas entre los lenguajes contemporáneos orientados a objetos— se centra en el tratamiento de las operaciones genéricas sobre tipos interrelacionados. Nuestra propia discusión sobre objetos computacionales en el Capítulo 3 evita por completo estos problemas. Los lectores familiarizados con la programación orientada a objetos notarán que tenemos mucho que decir en el capítulo 3 sobre el estado local, pero ni siquiera mencionamos "clases" o "herencia". De hecho, sospechamos que estos problemas no pueden abordarse adecuadamente en términos del diseño de lenguajes de computadora únicamente, sin recurrir también al trabajo en representación del conocimiento y razonamiento automatizado.

53. Un número real puede ser proyectado a un entero usando el primitivo `round`, que devuelve el entero más cercano a su argumento.

54. Por otro lado, permitiremos polinomios cuyos coeficientes sean a su vez polinomios en otras variables. Esto nos dará esencialmente el mismo poder de representación que un sistema multivariado completo, aunque conduce a problemas de coerción, como se discute más abajo.

55. Para polinomios univariados, dar el valor de un polinomio en un conjunto dado de puntos puede ser una representación particularmente buena. Esto hace que la aritmética de polinomios sea extremadamente simple. Para obtener, por ejemplo, la suma de dos polinomios representados de esta manera, solo necesitamos sumar los valores de los polinomios en los puntos correspondientes. Para transformar de nuevo a una representación más familiar, podemos usar la fórmula de interpolación de Lagrange, que muestra cómo recuperar los coeficientes de un polinomio de grado n dados los valores del polinomio en $n + 1$ puntos.

56. Esta operación es muy parecida a la operación ordenada `union/set` que desarrollamos en el Ejercicio 2.62. De hecho, si pensamos en los términos del polinomio como un conjunto ordenado según la potencia de la indeterminada, entonces el programa que produce la lista de términos para una suma es casi idéntico a `union/set`.

57. Para que esto funcione completamente sin problemas, también deberíamos agregar a nuestro sistema de aritmética genérica la capacidad de coaccionar un "número" a un polinomio considerándolo como un polinomio de grado cero cuyo coeficiente es el número. Esto es necesario si vamos a realizar operaciones como

$$[x^2 + (y + 1)x + 5] + [x^2 + 2x + 1],$$

lo que requiere sumar el coeficiente $y + 1$ al coeficiente 2.

58. En estos ejemplos de polinomios, asumimos que hemos implementado el sistema de aritmética genérica usando el mecanismo de tipo sugerido en el Ejercicio 2.78. Por lo tanto, los coeficientes que son números ordinarios estarán representados como los números mismos en lugar de como pares cuyo `car` es el símbolo `scheme/number`.

59. Aunque estamos asumiendo que las listas de términos están ordenadas, hemos implementado `adjoin/term` para simplemente hacer `cons` del nuevo término en la lista de términos existente. Podemos salirnos con la nuestra siempre que garanticemos que los procedimientos (como `add/terms`) que usan `adjoin/term` siempre lo llamen con un término de orden superior al que aparece en la lista. Si no quisiéramos hacer tal garantía, podríamos haber implementado `adjoin/term` para que fuera similar al constructor `adjoin/set` para la representación de lista ordenada de conjuntos (Ejercicio 2.61).

60. El hecho de que el Algoritmo de Euclides funcione para polinomios se formaliza en álgebra diciendo que los polinomios forman una especie de dominio algebraico llamado *anillo euclidiano*. Un anillo euclidiano es un dominio que admite suma, resta y multiplicación conmutativa, junto con una forma de asignar a cada elemento x del anillo una medida entera positiva $m(x)$ con las propiedades de que $m(xy) \geq m(x)$ para cualquier x y y no nulos y que, dado cualquier x y y, existe un q tal que $y = qx + r$ y o bien $r = 0$ o $m(r) < m(x)$. Desde un punto de vista abstracto, esto es lo que se necesita para probar que el Algoritmo de Euclides funciona. Para el dominio de los enteros, la medida m de un entero es el valor absoluto del entero mismo. Para el dominio de los polinomios, la medida de un polinomio es su grado.

61. En una implementación como MIT Scheme, esto produce un polinomio que es de hecho un divisor de Q_1 y Q_2, pero con coeficientes racionales. En muchos otros sistemas Scheme, en los que la división de enteros puede producir números decimales de precisión limitada, podríamos no obtener un divisor válido.

62. Un método extremadamente eficiente y elegante para calcular MCDs de polinomios fue descubierto por Richard Zippel (1979). El método es un algoritmo probabilístico, como lo es la prueba rápida de primalidad que discutimos en el Capítulo 1. El libro de Zippel (Zippel 1993) describe este método, junto con otras formas de calcular MCDs de polinomios.

Modularidad, Objetos y Estado

> Μεταβάλλον ἀναπαύεται
> (Incluso mientras cambia, permanece quieto.)
> —Heráclito
>
> Plus ça change, plus c'est la même chose.
> (Cuanto más cambia, más es lo mismo.)
> —Alphonse Karr

Los capítulos anteriores introdujeron los elementos básicos a partir de los cuales se construyen los programas. Vimos cómo los procedimientos primitivos y los datos primitivos se combinan para construir entidades compuestas, y aprendimos que la abstracción es vital para ayudarnos a lidiar con la complejidad de los grandes sistemas. Pero estas herramientas no son suficientes para diseñar programas. La síntesis eficaz de programas también requiere principios organizativos que puedan guiarnos en la formulación del diseño general de un programa. En particular, necesitamos estrategias para ayudarnos a estructurar sistemas grandes de modo que sean *modulares*, es decir, que puedan dividirse "naturalmente" en partes coherentes que puedan desarrollarse y mantenerse por separado.

Una poderosa estrategia de diseño, que es particularmente apropiada para la construcción de programas para modelar sistemas físicos, es basar la estructura de nuestros programas en la estructura del sistema que se está modelando. Para cada objeto en el sistema, construimos un objeto computacional correspondiente. Para cada acción del sistema, definimos una operación simbólica en nuestro modelo computacional. Nuestra esperanza al usar esta estrategia es que extender el modelo para acomodar nuevos objetos o nuevas acciones no requerirá cambios estratégicos en el programa, solo la adición de los nuevos análogos simbólicos de esos objetos o acciones. Si hemos tenido éxito en la organización de nuestro sistema, entonces para agregar una nueva característica o depurar una antigua tendremos que trabajar solo en una parte localizada del sistema.

En gran medida, entonces, la forma en que organizamos un programa grande está dictada por nuestra percepción del sistema a modelar. En este capítulo investigaremos dos estrategias organizativas prominentes que surgen de dos

"visiones del mundo" bastante diferentes sobre la estructura de los sistemas. La primera estrategia organizativa se concentra en los *objetos*, viendo un sistema grande como una colección de objetos distintos cuyos comportamientos pueden cambiar con el tiempo. Una estrategia organizativa alternativa se concentra en los *flujos* de información que fluyen en el sistema, de manera muy similar a como un ingeniero eléctrico ve un sistema de procesamiento de señales.

Tanto el enfoque basado en objetos como el enfoque de procesamiento de flujos plantean importantes cuestiones lingüísticas en la programación. Con los objetos, debemos preocuparnos por cómo un objeto computacional puede cambiar y, sin embargo, mantener su identidad. Esto nos obligará a abandonar nuestro antiguo modelo de sustitución de la computación (Section 1.1.5) en favor de un *modelo de entorno* de computación más mecanicista pero menos manejable teóricamente. Las dificultades de tratar con objetos, cambio e identidad son una consecuencia fundamental de la necesidad de lidiar con el tiempo en nuestros modelos computacionales. Estas dificultades se vuelven aún mayores cuando permitimos la posibilidad de ejecución concurrente de programas. El enfoque de flujos puede explotarse al máximo cuando desacoplamos el tiempo simulado en nuestro modelo del orden de los eventos que tienen lugar en la computadora durante la evaluación. Lograremos esto usando una técnica conocida como *evaluación retardada*.

Asignación y Estado Local

Normalmente vemos el mundo como poblado por objetos independientes, cada uno de los cuales tiene un estado que cambia con el tiempo. Se dice que un objeto "tiene estado" si su comportamiento está influenciado por su historia. Una cuenta bancaria, por ejemplo, tiene estado en el sentido de que la respuesta a la pregunta "¿Puedo retirar $100?" depende de la historia de las transacciones de depósito y retiro. Podemos caracterizar el estado de un objeto mediante una o más *variables de estado*, que entre ellas mantienen suficiente información sobre la historia para determinar el comportamiento actual del objeto. En un sistema bancario simple, podríamos caracterizar el estado de una cuenta por un saldo actual en lugar de recordar todo el historial de transacciones de la cuenta.

En un sistema compuesto por muchos objetos, los objetos rara vez son completamente independientes. Cada uno puede influir en los estados de otros a través de interacciones, que sirven para acoplar las variables de estado de un objeto a las de otros objetos. De hecho, la visión de que un sistema está compuesto por objetos separados es más útil cuando las

variables de estado del sistema pueden agruparse en subsistemas estrechamente acoplados que solo están débilmente acoplados a otros subsistemas.

Esta visión de un sistema puede ser un marco poderoso para organizar modelos computacionales del sistema. Para que tal modelo sea modular, debe descomponerse en objetos computacionales que modelen los objetos reales en el sistema. Cada objeto computacional debe tener sus propias *variables de estado locales* que describan el estado del objeto real. Dado que los estados de los objetos en el sistema que se está modelando cambian con el tiempo, las variables de estado de los correspondientes objetos computacionales también deben cambiar. Si elegimos modelar el flujo del tiempo en el sistema por el tiempo transcurrido en la computadora, entonces debemos tener una forma de construir objetos computacionales cuyos comportamientos cambien a medida que nuestros programas se ejecutan. En particular, si deseamos modelar variables de estado mediante nombres simbólicos ordinarios en el lenguaje de programación, entonces el lenguaje debe proporcionar un *operador de asignación* para permitirnos cambiar el valor asociado con un nombre.

Variables de Estado Locales

Para ilustrar lo que queremos decir con tener un objeto computacional con estado que varía en el tiempo, modelemos la situación de retirar dinero de una cuenta bancaria. Haremos esto usando un procedimiento `withdraw`, que toma como argumento una `cantidad` a retirar. Si hay suficiente dinero en la cuenta para acomodar el retiro, entonces `withdraw` debería devolver el saldo restante después del retiro. De lo contrario, `withdraw` debería devolver el mensaje *Fondos insuficientes*. Por ejemplo, si comenzamos con $100 en la cuenta, deberíamos obtener la siguiente secuencia de respuestas usando `withdraw`:

(withdraw 25) *75* (withdraw 25) *50* (withdraw 60) *"Fondos insuficientes"* (withdraw 15) *35*

Observe que la expresión `(withdraw 25)`, evaluada dos veces, produce valores diferentes. Este es un nuevo tipo de comportamiento para un procedimiento. Hasta ahora, todos nuestros procedimientos podían verse como especificaciones para calcular funciones matemáticas. Una llamada a un procedimiento calculaba el valor de la función aplicada a los argumentos dados, y dos llamadas al mismo procedimiento con los mismos argumentos siempre producían el mismo resultado.[1]

Para implementar `withdraw`, podemos usar una variable `balance` para

indicar el saldo de dinero en la cuenta y definir `withdraw` como un procedimiento que accede a `balance`. El procedimiento `withdraw` comprueba si `balance` es al menos tan grande como la `cantidad` solicitada. Si es así, `withdraw` decrementa `balance` por la `cantidad` y devuelve el nuevo valor de `balance`. De lo contrario, `withdraw` devuelve el mensaje *Fondos insuficientes*. Aquí están las definiciones de `balance` y `withdraw`:

(define balance 100) (define (withdraw amount) (if (>= balance amount) (begin (set! balance (- balance amount)) balance) "Fondos insuficientes"))

Decrementar `balance` se logra mediante la expresión

(set! balance (- balance amount))

Esto usa la forma especial `set!`, cuya sintaxis es

(set! ⟨ *nombre* ⟩ ⟨ *nuevo-valor* ⟩)

Aquí ⟨*nombre*$\kern0.04em\rangle$ es un símbolo y ⟨*nuevo-valor*$\kern0.04em\rangle$ es cualquier expresión. `set!` cambia ⟨*nombre*$\kern0.04em\rangle$ de modo que su valor sea el resultado obtenido al evaluar ⟨*nuevo-valor*$\kern0.04em\rangle$. En el caso que nos ocupa, estamos cambiando `balance` de modo que su nuevo valor será el resultado de restar `amount` del valor anterior de `balance`.[2]

`withdraw` también usa la forma especial `begin` para hacer que se evalúen dos expresiones en el caso de que la prueba `if` sea verdadera: primero decrementando `balance` y luego devolviendo el valor de `balance`. En general, evaluar la expresión

(begin ⟨ *exp* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$ ⟨ *exp* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$ … ⟨ *exp* $\color{SchemeDark}_{\hbox{\ttfamily\itshape\scriptsize k}}\rangle$)

hace que las expresiones $\langle\kern0.06em$*exp*$_1$⟩ a $\langle\kern0.06em$*expk*⟩ se evalúen en secuencia y que el valor de la expresión final $\langle\kern0.06em$*expk*⟩ se devuelva como el valor de toda la forma `begin`.[3]

Aunque `withdraw` funciona como se desea, la variable `balance` presenta un problema. Como se especificó anteriormente, `balance` es un nombre definido en el entorno global y es libremente accesible para ser examinado o modificado por cualquier procedimiento. Sería mucho mejor si pudiéramos de alguna manera hacer que `balance` sea interno a `withdraw`, de modo que `withdraw` fuera el único procedimiento que pudiera acceder a `balance`

directamente y cualquier otro procedimiento pudiera acceder a `balance` solo indirectamente (a través de llamadas a `withdraw`). Esto modelaría con mayor precisión la noción de que `balance` es una variable de estado local utilizada por `withdraw` para realizar un seguimiento del estado de la cuenta.

Podemos hacer que `balance` sea interno a `withdraw` reescribiendo la definición de la siguiente manera:

(define new-withdraw (let ((balance 100)) (lambda (amount) (if (> = balance amount) (begin (set! balance (- balance amount)) balance) "Fondos insuficientes"))))

Lo que hemos hecho aquí es usar `let` para establecer un entorno con una variable local `balance`, ligada al valor inicial 100. Dentro de este entorno local, usamos `lambda` para crear un procedimiento que toma `amount` como argumento y se comporta como nuestro procedimiento `withdraw` anterior. Este procedimiento—devuelto como el resultado de evaluar la expresión `let` —es `new/withdraw`, que se comporta precisamente de la misma manera que `withdraw` pero cuya variable `balance` no es accesible por ningún otro procedimiento.[4]

Combinar `set!` con variables locales es la técnica de programación general que usaremos para construir objetos computacionales con estado local. Desafortunadamente, usar esta técnica plantea un problema serio: cuando introdujimos por primera vez los procedimientos, también introdujimos el modelo de evaluación por sustitución (Section 1.1.5) para proporcionar una interpretación de lo que significa la aplicación de procedimientos. Dijimos que aplicar un procedimiento debería interpretarse como evaluar el cuerpo del procedimiento con los parámetros formales reemplazados por sus valores. El problema es que, tan pronto como introducimos la asignación en nuestro lenguaje, la sustitución ya no es un modelo adecuado de aplicación de procedimientos. (Veremos por qué esto es así en Section 3.1.3.) Como consecuencia, técnicamente no tenemos en este momento ninguna forma de entender por qué el procedimiento `new/withdraw` se comporta como se afirmó anteriormente. Para entender realmente un procedimiento como `new/withdraw`, necesitaremos desarrollar un nuevo modelo de aplicación de procedimientos. En Section 3.2 introduciremos tal modelo, junto con una explicación de `set!` y las variables locales. Primero, sin embargo, examinamos algunas variaciones sobre el tema establecido por `new/withdraw`.

El siguiente procedimiento, `make/withdraw`, crea "procesadores de retiro". El parámetro formal `balance` en `make/withdraw` especifica la cantidad inicial de dinero en la cuenta.[5]

(define (make-withdraw balance) (lambda (amount) (if (>= balance amount) (begin (set! balance (- balance amount)) balance) "Fondos insuficientes")))

`make/withdraw` se puede usar de la siguiente manera para crear dos objetos `W1` y `W2`:

(define W1 (make-withdraw 100)) (define W2 (make-withdraw 100))

(W1 50) *50* (W2 70) *30* (W2 40) *"Fondos insuficientes"* (W1 40) *10*

Observe que `W1` y `W2` son objetos completamente independientes, cada uno con su propia variable de estado local `balance`. Los retiros de uno no afectan al otro.

También podemos crear objetos que manejan depósitos además de retiros, y así podemos representar cuentas bancarias simples. Aquí hay un procedimiento que devuelve un "objeto de cuenta bancaria" con un saldo inicial especificado:

(define (make-account balance) (define (withdraw amount) (if (>= balance amount) (begin (set! balance (- balance amount)) balance) "Fondos insuficientes")) (define (deposit amount) (set! balance (+ balance amount)) balance) (define (dispatch m) (cond ((eq? m 'withdraw) withdraw) ((eq? m 'deposit) deposit) (else (error "Solicitud desconocida: MAKE-ACCOUNT" m)))) dispatch)

Cada llamada a `make/account` establece un entorno con una variable de estado local `balance`. Dentro de este entorno, `make/account` define los procedimientos `deposit` y `withdraw` que acceden a `balance` y un procedimiento adicional `dispatch` que toma un "mensaje" como entrada y devuelve uno de los dos procedimientos locales. El procedimiento `dispatch` mismo se devuelve como el valor que representa el objeto de cuenta bancaria. Este es precisamente el estilo de programación de *paso de mensajes* que vimos en Section 2.4.3, aunque aquí lo estamos usando junto con la capacidad de modificar variables locales.

`make/account` se puede usar de la siguiente manera:

(define acc (make-account 100)) ((acc 'withdraw) 50) *50* ((acc 'withdraw) 60) *"Fondos insuficientes"* ((acc 'deposit) 40) *90* ((acc 'withdraw) 60) *30*

Cada llamada a `acc` devuelve el procedimiento `deposit` o `withdraw` definido localmente, que luego se aplica a la `cantidad` especificada. Como fue el caso con `make/withdraw`, otra llamada a `make/account`

(define acc2 (make-account 100))

producirá un objeto de cuenta completamente separado, que mantiene su propio `balance` local.

Ejercicio 3.1: Un *acumulador* es un procedimiento que se llama repetidamente con un solo argumento numérico y acumula sus argumentos en una suma. Cada vez que se llama, devuelve la suma acumulada actualmente. Escribe un procedimiento `make/accumulator` que genere acumuladores, cada uno manteniendo una suma independiente. La entrada a `make/accumulator` debería especificar el valor inicial de la suma; por ejemplo

(define A (make-accumulator 5)) (A 10) *15* (A 10) *25*

Ejercicio 3.2: En aplicaciones de prueba de software, es útil poder contar el número de veces que se llama a un procedimiento dado durante el transcurso de un cálculo. Escribe un procedimiento `make/monitored` que tome como entrada un procedimiento, `f`, que a su vez toma una entrada. El resultado devuelto por `make/monitored` es un tercer procedimiento, digamos `mf`, que lleva la cuenta del número de veces que ha sido llamado manteniendo un contador interno. Si la entrada a `mf` es el símbolo especial `how/many/calls?`, entonces `mf` devuelve el valor del contador. Si la entrada es el símbolo especial `reset/count`, entonces `mf` restablece el contador a cero. Para cualquier otra entrada, `mf` devuelve el resultado de llamar a `f` con esa entrada e incrementa el contador. Por ejemplo, podríamos hacer una versión monitoreada del procedimiento `sqrt`:

(define s (make-monitored sqrt)) (s 100) *10* (s 'how-many-calls?) *1*

Ejercicio 3.3: Modifica el procedimiento `make/account` para que cree cuentas protegidas por contraseña. Es decir, `make/account` debería tomar un símbolo como argumento adicional, como en

(define acc (make-account 100 'secret-password))

El objeto de cuenta resultante debería procesar una solicitud solo si va acompañada de la contraseña con la que se creó la cuenta, y de lo contrario debería devolver una queja:

((acc 'secret-password 'withdraw) 40) *60* ((acc 'some-other-password 'deposit) 50) *"Contraseña incorrecta"*

Ejercicio 3.4: Modifica el procedimiento `make/account` del Ejercicio 3.3 agregando otra variable de estado local para que, si se accede a una

cuenta más de siete veces consecutivas con una contraseña incorrecta, invoque el procedimiento `call/the/cops`.

Los Beneficios de Introducir la Asignación

Como veremos, introducir la asignación en nuestro lenguaje de programación nos lleva a un laberinto de difíciles cuestiones conceptuales. Sin embargo, ver los sistemas como colecciones de objetos con estado local es una técnica poderosa para mantener un diseño modular. Como ejemplo simple, consideremos el diseño de un procedimiento `rand` que, cada vez que se llama, devuelve un entero elegido al azar.

No está del todo claro qué se entiende por "elegido al azar". Lo que presumiblemente queremos es que las sucesivas llamadas a `rand` produzcan una secuencia de números que tenga las propiedades estadísticas de una distribución uniforme. No discutiremos aquí los métodos para generar secuencias adecuadas. Más bien, supongamos que tenemos un procedimiento `rand/update` que tiene la propiedad de que si comenzamos con un número dado x_1 y formamos

$$x_2 = (\text{rand-update } x_1) \quad x_3 = (\text{rand-update } x_2)$$

entonces la secuencia de valores x_1, x_2, x_3, … tendrá las propiedades estadísticas deseadas.[6]

Podemos implementar `rand` como un procedimiento con una variable de estado local `x` que se inicializa a algún valor fijo `random/init`. Cada llamada a `rand` calcula `rand/update` del valor actual de `x`, devuelve esto como el número aleatorio, y también almacena esto como el nuevo valor de `x`.

```
(define rand (let ((x random-init)) (lambda () (set! x (rand-update x)) x)))
```

Por supuesto, podríamos generar la misma secuencia de números aleatorios sin usar la asignación simplemente llamando a `rand/update` directamente. Sin em-bargo, esto significaría que cualquier parte de nuestro programa que usara números aleatorios tendría que recordar explícitamente el valor actual de `x` para pasarlo como argumento a `rand/update`. Para darse cuenta de lo molesto que sería esto, considere usar números aleatorios para implementar una técnica llamada *simulación de Monte Carlo*.

El método de Monte Carlo consiste en elegir experimentos de muestra al azar de un conjunto grande y luego hacer deducciones sobre la base de las probabilidades estimadas a partir de la tabulación de los resultados de esos experimentos. Por ejemplo, podemos aproximar π usando el hecho de que 6/

π^2 es la probabilidad de que dos enteros elegidos al azar no tengan factores en común; es decir, que su máximo común divisor sea 1.[7] Para obtener la aproximación a π, realizamos un gran número de experimentos. En cada experimento elegimos dos enteros al azar y realizamos una prueba para ver si su MCD es 1. La fracción de veces que la prueba es superada nos da nuestra estimación de $6/\pi^2$, y de esto obtenemos nuestra aproximación a π.

El corazón de nuestro programa es un procedimiento `monte/carlo`, que toma como argumentos el número de veces que se debe intentar un experimento, junto con el experimento, representado como un procedimiento sin argumentos que devolverá verdadero o falso cada vez que se ejecute. `monte/carlo` ejecuta el experimento durante el número designado de pruebas y devuelve un número que indica la fracción de las pruebas en las que el experimento se encontró que era verdadero.

(define (estimate-pi trials) (sqrt (/ 6 (monte-carlo trials cesaro-test)))) (define (cesaro-test) (= (gcd (rand) (rand)) 1))

(define (monte-carlo trials experiment) (define (iter trials-remaining trials-passed) (cond ((= trials-remaining 0) (/ trials-passed trials)) ((experiment) (iter (- trials-remaining 1) (+ trials-passed 1))) (else (iter (- trials-remaining 1) trials-passed)))) (iter trials 0))

Ahora intentemos el mismo cálculo usando `rand/update` directamente en lugar de `rand`, la forma en que nos veríamos forzados a proceder si no usáramos la asignación para modelar el estado local:

(define (estimate-pi trials) (sqrt (/ 6 (random-gcd-test trials random-init)))) (define (random-gcd-test trials initial-x) (define (iter trials-remaining trials-passed x) (let ((x1 (rand-update x))) (let ((x2 (rand-update x1))) (cond ((= trials-remaining 0) (/ trials-passed trials)) ((= (gcd x1 x2) 1) (iter (- trials-remaining 1) (+ trials-passed 1) x2)) (else (iter (- trials-remaining 1) trials-passed x2)))))) (iter trials 0 initial-x))

Aunque el programa sigue siendo simple, delata algunas dolorosas brechas en la modularidad. En nuestra primera versión del programa, usando `rand`, podemos expresar el método de Monte Carlo directamente como un procedimiento general `monte/carlo` que toma como argumento un procedimiento `experiment` arbitrario. En nuestra segunda versión del programa, sin estado local para el generador de números aleatorios, `random/gcd/test` debe manipular explícitamente los números aleatorios `x1` y `x2` y reciclar `x2` a través del bucle iterativo como la nueva entrada para `rand/update`. Este manejo explícito de los números aleatorios entrelaza la estructura de la acumulación de resultados de la prueba con el hecho de que

nuestro experimento particular usa dos números aleatorios, mientras que otros experimentos de Monte Carlo podrían usar un número aleatorio o tres. Incluso el procedimiento de nivel superior `estimate/pi` tiene que preocuparse por proporcionar un número aleatorio inicial. El hecho de que las interioridades del generador de números aleatorios se estén filtrando a otras partes del programa nos dificulta aislar la idea de Monte Carlo para que pueda aplicarse a otras tareas. En la primera versión del programa, la asignación encapsula el estado del generador de números aleatorios dentro del procedimiento `rand`, de modo que los detalles de la generación de números aleatorios permanecen independientes del resto del programa.

El fenómeno general ilustrado por el ejemplo de Monte Carlo es este: Desde el punto de vista de una parte de un proceso complejo, las otras partes parecen cambiar con el tiempo. Tienen un estado local oculto que varía en el tiempo. Si deseamos escribir programas de computadora cuya estructura refleje esta descomposición, hacemos objetos computacionales (como cuentas bancarias y generadores de números aleatorios) cuyo comportamiento cambia con el tiempo. Modelamos el estado con variables de estado locales, y modelamos los cambios de estado con asignaciones a esas variables.

Es tentador concluir esta discusión diciendo que, al introducir la asignación y la técnica de ocultar el estado en variables locales, somos capaces de estructurar los sistemas de una manera más modular que si todo el estado tuviera que ser manipulado explícitamente, pasando parámetros adicionales. Desafortunadamente, como veremos, la historia no es tan simple.

> **Ejercicio 3.5:** La *integración de Monte Carlo* es un método para estimar integrales definidas por medio de la simulación de Monte Carlo. Considere calcular el área de una región del espacio descrita por un predicado $P(x,y)$ que es verdadero para los puntos (x,y) en la región y falso para los puntos que no están en la región. Por ejemplo, la región contenida dentro de un círculo de radio 3 centrado en (5, 7) se describe por el predicado que prueba si $(x-5)^2 + (y-7)^2 \leq 3^2$. Para estimar el área de la región descrita por tal predicado, comience por elegir un rectángulo que contenga la región. Por ejemplo, un rectángulo con esquinas diagonalmente opuestas en (2, 4) y (8, 10) contiene el círculo anterior. La integral deseada es el área de esa porción del rectángulo que se encuentra en la región. Podemos estimar la integral eligiendo, al azar, puntos (x,y) que se encuentran en el rectángulo, y probando $P(x,y)$ para cada punto para determinar si el punto se encuentra en la región. Si intentamos esto con muchos puntos, entonces la fracción de puntos que caen en la región debería dar una estimación de la proporción del rectángulo que se encuentra en la región. Por lo tanto, multiplicar esta fracción por el área de todo el rectángulo debería producir una

estimación de la integral.

Implemente la integración de Monte Carlo como un procedimiento `estimate/integral` que tome como argumentos un predicado `P`, límites superior e inferior `x1`, `x2`, `y1`, y `y2` para el rectángulo, y el número de pruebas a realizar para producir la estimación. Su procedimiento debería usar el mismo procedimiento `monte/carlo` que se usó anteriormente para estimar π. Use su `estimate/integral` para producir una estimación de π midiendo el área de un círculo unitario.

Le resultará útil tener un procedimiento que devuelva un número elegido al azar de un rango dado. El siguiente procedimiento `random/in/range` implementa esto en términos del procedimiento `random` usado en Section 1.2.6, que devuelve un número no negativo menor que su entrada.[8]

(define (random-in-range low high) (let ((range (- high low))) (+ low (random range))))

Ejercicio 3.6: Es útil poder reiniciar un generador de números aleatorios para producir una secuencia que comience desde un valor dado. Diseñe un nuevo procedimiento `rand` que se llame con un argumento que sea o el símbolo `generate` o el símbolo `reset` y se comporte de la siguiente manera: `(rand 'generate)` produce un nuevo número aleatorio; `((rand 'reset)` ⟨*nuevo-valor*$\kern0.11em\rangle$`)` reinicia la variable de estado interna al ⟨*nuevo-valor*$\kern0.08em\rangle$ designado. Así, al reiniciar el estado, se pueden generar secuencias repetibles. Estas son muy útiles para tener al probar y depurar programas que usan números aleatorios.

Los Costos de Introducir la Asignación

Como hemos visto, la operación `set!` nos permite modelar objetos que tienen estado local. Sin embargo, esta ventaja tiene un precio. Nuestro lenguaje de programación ya no puede interpretarse en términos del modelo de sustitución de la aplicación de procedimientos que introdujimos en Section 1.1.5. Además, ningún modelo simple con propiedades matemáticas "agradables" puede ser un marco adecuado para tratar con objetos y asignación en lenguajes de programación.

Mientras no usemos asignaciones, dos evaluaciones del mismo procedimiento con los mismos argumentos producirán el mismo resultado, de modo que los procedimientos pueden verse como el cálculo de funciones matemáticas. La programación sin ningún uso de asignaciones, como hicimos a lo largo de los dos primeros capítulos de este libro, se conoce en consecuencia como

programación funcional.

Para entender cómo la asignación complica las cosas, considere una versión simplificada del procedimiento `make/withdraw` de Section 3.1.1 que no se molesta en comprobar si hay una cantidad insuficiente:

(define (make-simplified-withdraw balance) (lambda (amount) (set! balance (- balance amount)) balance)) (define W (make-simplified-withdraw 25)) (W 20) *5* (W 10) *-5*

Compare este procedimiento con el siguiente procedimiento `make/decrementer`, que no usa `set!`:

(define (make-decrementer balance) (lambda (amount) (- balance amount)))

`make/decrementer` devuelve un procedimiento que resta su entrada de una cantidad designada `balance`, pero no hay un efecto acumulado sobre llamadas sucesivas, como con `make/simplified/withdraw`:

(define D (make-decrementer 25)) (D 20) *5* (D 10) *15*

Podemos usar el modelo de sustitución para explicar cómo funciona `make/decrementer`. Por ejemplo, analicemos la evaluación de la expresión

((make-decrementer 25) 20)

Primero simplificamos el operador de la combinación sustituyendo 25 por `balance` en el cuerpo de `make/decrementer`. Esto reduce la expresión a

((lambda (amount) (- 25 amount)) 20)

Ahora aplicamos el operador sustituyendo 20 por `amount` en el cuerpo de la expresión `lambda`:

(- 25 20)

La respuesta final es 5.

Observe, sin embargo, lo que sucede si intentamos un análisis de sustitución similar con `make/simplified/withdraw`:

((make-simplified-withdraw 25) 20)

Primero simplificamos el operador sustituyendo 25 por `balance` en el cuerpo de `make/simplified/withdraw`. Esto reduce la expresión a[9]

((lambda (amount) (set! balance (- 25 amount)) 25) 20)

Ahora aplicamos el operador sustituyendo 20 por `amount` en el cuerpo de la expresión `lambda`:

(set! balance (- 25 20)) 25

Si nos adhiriéramos al modelo de sustitución, tendríamos que decir que el significado de la aplicación del procedimiento es primero establecer `balance` en 5 y luego devolver 25 como el valor de la expresión. Esto obtiene la respuesta incorrecta. Para obtener la respuesta correcta, tendríamos que distinguir de alguna manera la primera ocurrencia de `balance` (antes del efecto del `set!`) de la segunda ocurrencia de `balance` (después del efecto del `set!`), y el modelo de sustitución no puede hacer esto.

El problema aquí es que la sustitución se basa en última instancia en la noción de que los símbolos en nuestro lenguaje son esencialmente nombres para valores. Pero tan pronto como introducimos `set!` y la idea de que el valor de una variable puede cambiar, una variable ya no puede ser simplemente un nombre. Ahora una variable de alguna manera se refiere a un lugar donde se puede almacenar un valor, y el valor almacenado en este lugar puede cambiar. En Section 3.2 veremos cómo los entornos desempeñan este papel de "lugar" en nuestro modelo computacional.

Identidad y cambio

El problema que surge aquí es más profundo que la mera ruptura de un modelo particular de computación. Tan pronto como introducimos el cambio en nuestros modelos computacionales, muchas nociones que antes eran sencillas se vuelven problemáticas. Considere el concepto de que dos cosas son "lo mismo".

Supongamos que llamamos a `make/decrementer` dos veces con el mismo argumento para crear dos procedimientos:

(define D1 (make-decrementer 25)) (define D2 (make-decrementer 25))

¿Son `D1` y `D2` lo mismo? Una respuesta aceptable es sí, porque `D1` y `D2` tienen el mismo comportamiento computacional—cada uno es un procedimiento que resta su entrada de 25. De hecho, `D1` podría ser sustituido por `D2` en cualquier cálculo sin cambiar el resultado.

Contraste esto con hacer dos llamadas a `make/simplified/withdraw`:

(define W1 (make-simplified-withdraw 25)) (define W2 (make-simplified-

withdraw 25))

¿Son `W1` y `W2` lo mismo? Seguramente no, porque las llamadas a `W1` y `W2` tienen efectos distintos, como se muestra en la siguiente secuencia de interacciones:

(W1 20) *5* (W1 20) *-15* (W2 20) *5*

Aunque `W1` y `W2` son "iguales" en el sentido de que ambos se crean evaluando la misma expresión, `(make/simplified/withdraw 25)`, no es cierto que `W1` pueda ser sustituido por `W2` en cualquier expresión sin cambiar el resultado de evaluar la expresión.

Se dice que un lenguaje que apoya el concepto de que "iguales pueden ser sustituidos por iguales" en una expresión sin cambiar el valor de la expresión es *referencialmente transparente*. La transparencia referencial se viola cuando incluimos `set!` en nuestro lenguaje de computadora. Esto hace que sea difícil determinar cuándo podemos simplificar expresiones sustituyendo expresiones equivalentes. En consecuencia, razonar sobre programas que usan asignación se vuelve drásticamente más difícil.

Una vez que renunciamos a la transparencia referencial, la noción de lo que significa que los objetos computacionales sean "lo mismo" se vuelve difícil de capturar de una manera formal. De hecho, el significado de "mismo" en el mundo real que nuestros programas modelan apenas está claro en sí mismo. En general, podemos determinar que dos objetos aparentemente idénticos son de hecho "el mismo" solo modificando un objeto y luego observando si el otro objeto ha cambiado de la misma manera. Pero, ¿cómo podemos saber si un objeto ha "cambiado" si no es observando el "mismo" objeto dos veces y viendo si alguna propiedad del objeto difiere de una observación a la siguiente? Por lo tanto, no podemos determinar el "cambio" sin alguna noción *a priori* de "identidad", y no podemos determinar la identidad sin observar los efectos del cambio.

Como ejemplo de cómo surge este problema en la programación, considere la situación donde Peter y Paul tienen una cuenta bancaria con $100. Hay una diferencia sustancial entre modelar esto como

(define peter-acc (make-account 100)) (define paul-acc (make-account 100))

y modelarlo como

(define peter-acc (make-account 100)) (define paul-acc peter-acc)

En la primera situación, las dos cuentas bancarias son distintas. Las

transacciones hechas por Peter no afectarán la cuenta de Paul, y viceversa. En la segunda situación, sin embargo, hemos definido `paul/acc` como *lo mismo* que `peter/acc`. En efecto, Peter y Paul ahora tienen una cuenta bancaria conjunta, y si Peter hace un retiro de `peter/acc`, Paul observará menos dinero en `paul/acc`. Estas dos situaciones similares pero distintas pueden causar confusión al construir modelos computacionales. Con la cuenta compartida, en particular, puede ser especialmente confuso que haya un objeto (la cuenta bancaria) que tiene dos nombres diferentes (`peter/acc` y `paul/acc`); si estamos buscando todos los lugares en nuestro programa donde `paul/acc` puede ser cambiado, debemos recordar mirar también las cosas que cambian `peter/acc`.[10]

Con referencia a las observaciones anteriores sobre "identidad" y "cambio", observe que si Peter y Paul solo pudieran examinar sus saldos bancarios, y no pudieran realizar operaciones que cambiaran el saldo, entonces el problema de si las dos cuentas son distintas sería irrelevante. En general, mientras nunca modifiquemos objetos de datos, podemos considerar que un objeto de datos compuesto es precisamente la totalidad de sus piezas. Por ejemplo, un número racional se determina dando su numerador y su denominador. Pero esta visión ya no es válida en presencia de cambio, donde un objeto de datos compuesto tiene una "identidad" que es algo diferente de las piezas de las que está compuesto. Una cuenta bancaria sigue siendo "la misma" cuenta bancaria incluso si cambiamos el saldo haciendo un retiro; a la inversa, podríamos tener dos cuentas bancarias diferentes con la misma información de estado. Esta complicación es una consecuencia, no de nuestro lenguaje de programación, sino de nuestra percepción de una cuenta bancaria como un objeto. No consideramos, por ejemplo, ordinariamente un número racional como un objeto cambiable con identidad, tal que pudiéramos cambiar el numerador y seguir teniendo "el mismo" número racional.

Peligros de la programación imperativa

En contraste con la programación funcional, la programación que hace un uso extensivo de la asignación se conoce como *programación imperativa*. Además de plantear complicaciones sobre los modelos computacionales, los programas escritos en estilo imperativo son susceptibles a errores que no pueden ocurrir en los programas funcionales. Por ejemplo, recuerde el programa factorial iterativo de la Sección 1.2.1:

(define (factorial n) (define (iter product counter) (if (> counter n) product (iter (* counter product) (+ counter 1)))) (iter 1 1))

En lugar de pasar argumentos en el bucle iterativo interno, podríamos adoptar un estilo más imperativo usando asignación explícita para actualizar

los valores de las variables `product` y `counter`:

(define (factorial n) (let ((product 1) (counter 1)) (define (iter) (if (> counter n) product (begin (set! product (* counter product)) (set! counter (+ counter 1)) (iter)))) (iter)))

Esto no cambia los resultados producidos por el programa, pero introduce una trampa sutil. ¿Cómo decidimos el orden de las asignaciones? Resulta que el programa es correcto como está escrito. Pero escribir las asignaciones en el orden opuesto

(set! counter (+ counter 1)) (set! product (* counter product))

habría producido un resultado diferente e incorrecto. En general, la programación con asignación nos obliga a considerar cuidadosamente los órdenes relativos de las asignaciones para asegurarnos de que cada declaración esté usando la versión correcta de las variables que han sido cambiadas. Este problema simplemente no surge en los programas funcionales.[11]

La complejidad de los programas imperativos empeora aún más si consideramos aplicaciones en las que varios procesos se ejecutan simultáneamente. Volveremos a esto en Section 3.4. Primero, sin embargo, abordaremos el problema de proporcionar un modelo computacional para expresiones que involucran asignación, y exploraremos los usos de los objetos con estado local en el diseño de simulaciones.

> **Ejercicio 3.7:** Considere los objetos de cuenta bancaria creados por `make/account`, con la modificación de contraseña descrita en el Ejercicio 3.3. Suponga que nuestro sistema bancario requiere la capacidad de crear cuentas conjuntas. Defina un procedimiento `make/joint` que logre esto. `make/joint` debe tomar tres argumentos. El primero es una cuenta protegida por contraseña. El segundo argumento debe coincidir con la contraseña con la que se definió la cuenta para que la operación `make/joint` proceda. El tercer argumento es una nueva contraseña. `make/joint` debe crear un acceso adicional a la cuenta original usando la nueva contraseña. Por ejemplo, si `peter/acc` es una cuenta bancaria con contraseña `open/sesame`, entonces
>
> (define paul-acc (make-joint peter-acc 'open-sesame 'rosebud))
>
> permitirá realizar transacciones en `peter/acc` usando el nombre `paul/acc` y la contraseña `rosebud`. Es posible que desee modificar su solución al Ejercicio 3.3 para acomodar esta nueva característica.

Ejercicio 3.8: Cuando definimos el modelo de evaluación en Section 1.1.3, dijimos que el primer paso para evaluar una expresión es evaluar sus subexpresiones. Pero nunca especificamos el orden en que las subexpresiones deberían evaluarse (por ejemplo, de izquierda a derecha o de derecha a izquierda). Cuando introducimos la asignación, el orden en que se evalúan los argumentos de un procedimiento puede marcar una diferencia en el resultado. Defina un procedimiento simple `f` tal que evaluar

```
(+ (f 0) (f 1))
```

devolverá 0 si los argumentos de `+` se evalúan de izquierda a derecha, pero devolverá 1 si los argumentos se evalúan de derecha a izquierda.

El Modelo de Entorno de Evaluación

Cuando introdujimos los procedimientos compuestos en el Capítulo 1, usamos el modelo de evaluación por sustitución (Section 1.1.5) para definir lo que significa aplicar un procedimiento a argumentos:

- Para aplicar un procedimiento compuesto a argumentos, evalúe el cuerpo del procedimiento con cada parámetro formal reemplazado por el argumento correspondiente.

Una vez que admitimos la asignación en nuestro lenguaje de programación, tal definición ya no es adecuada. En particular, Section 3.1.3 argumentó que, en presencia de asignación, una variable ya no puede considerarse simplemente como un nombre para un valor. Más bien, una variable debe designar de alguna manera un "lugar" en el que se pueden almacenar valores. En nuestro nuevo modelo de evaluación, estos lugares se mantendrán en estructuras llamadas *entornos*.

Un entorno es una secuencia de *marcos*. Cada marco es una tabla (posiblemente vacía) de *ligaduras*, que asocian nombres de variables con sus valores correspondientes. (Un solo marco puede contener como máximo una ligadura para cualquier variable.) Cada marco también tiene un puntero a su *entorno envolvente*, a menos que, para fines de discusión, el marco se considere *global*. El *valor de una variable* con respecto a un entorno es el valor dado por la ligadura de la variable en el primer marco en el entorno que contiene una ligadura para esa variable. Si ningún marco en la secuencia especifica una ligadura para la variable, entonces se dice que la variable está *no ligada* en el entorno.

La Figura 3.1 muestra una estructura de entorno simple que consta de tres marcos, etiquetados I, II y III. En el diagrama, A, B, C y D son punteros a entornos. C y D apuntan al mismo entorno. Las variables `z` y `x` están ligadas en el marco II, mientras que `y` y `x` están ligadas en el marco I. El valor de `x` en el entorno D es 3. El valor de `x` con respecto al entorno B también es 3. Esto se determina de la siguiente manera: Examinamos el primer marco en la secuencia (marco III) y no encontramos una ligadura para `x`, por lo que procedemos al entorno envolvente D y encontramos la ligadura en el marco I. Por otro lado, el valor de `x` en el entorno A es 7, porque el primer marco en la secuencia (marco II) contiene una ligadura de `x` a 7. Con respecto al entorno A, se dice que la ligadura de `x` a 7 en el marco II *sombrea* la ligadura de `x` a 3 en el marco I.

Figura 3.1: Una estructura de entorno simple.

El entorno es crucial para el proceso de evaluación, porque determina el contexto en el que debe evaluarse una expresión. De hecho, se podría decir que las expresiones en un lenguaje de programación no tienen, por sí mismas, ningún significado. Más bien, una expresión adquiere un significado solo con respecto a algún entorno en el que se evalúa. Incluso la interpretación de una expresión tan sencilla como `(+ 1 1)` depende de la comprensión de que se está operando en un contexto en el que `+` es el símbolo para la suma. Así, en nuestro modelo de evaluación siempre hablaremos de evaluar una expresión con respecto a algún entorno. Para describir las interacciones con el intérprete, supondremos que hay un entorno global, que consta de un único marco (sin entorno envolvente) que incluye valores para los símbolos asociados con los procedimientos primitivos. Por ejemplo, la idea de que `+` es el símbolo de la adición se captura diciendo que el símbolo `+` está ligado en el entorno global al procedimiento de adición primitivo.

Las Reglas para la Evaluación

La especificación general de cómo el intérprete evalúa una combinación sigue siendo la misma que cuando la introdujimos por primera vez en Section 1.1.3:

- Para evaluar una combinación:

1. Evaluar las subexpresiones de la combinación.[12]
2. Aplicar el valor de la subexpresión del operador a los valores de las subexpresiones del operando.

El modelo de evaluación de entorno reemplaza al modelo de sustitución al especificar lo que significa aplicar un procedimiento compuesto a argumentos.

En el modelo de evaluación de entorno, un procedimiento es siempre un par que consiste en algún código y un puntero a un entorno. Los procedimientos se crean de una sola manera: evaluando una expresión λ. Esto produce un procedimiento cuyo código se obtiene del texto de la expresión λ y cuyo entorno es el entorno en el que se evaluó la expresión λ para producir el procedimiento. Por ejemplo, considere la definición del procedimiento

(define (square x) (* x x))

evaluado en el entorno global. La sintaxis de definición de procedimiento es solo azúcar sintáctico para una expresión λ implícita subyacente. Hubiera sido equivalente haber usado

(define square (lambda (x) (* x x)))

que evalúa `(lambda (x) (* x x))` y liga `square` al valor resultante, todo en el entorno global.

La Figura 3.2 muestra el resultado de evaluar esta expresión `define`. El objeto de procedimiento es un par cuyo código especifica que el procedimiento tiene un parámetro formal, a saber, `x`, y un cuerpo de procedimiento `(* x x)`. La parte de entorno del procedimiento es un puntero al entorno global, ya que ese es el entorno en el que se evaluó la expresión λ para producir el procedimiento. Se ha agregado una nueva ligadura, que asocia el objeto de procedimiento con el símbolo `square`, al marco global. En general, `define` crea definiciones agregando ligaduras a los marcos.

Figura 3.2: Estructura de entorno producida al evaluar `(define (square x) (* x x))` en el entorno global.

Ahora que hemos visto cómo se crean los procedimientos, podemos describir cómo se aplican los procedimientos. El modelo de entorno especifica: Para aplicar un procedimiento a argumentos, cree un nuevo entorno que contenga un marco que ligue los parámetros a los valores de los argumentos. El entorno envolvente de este marco es el entorno especificado por el procedimiento. Ahora, dentro de este nuevo entorno, evalúe el cuerpo del procedimiento.

Para mostrar cómo se sigue esta regla, la Figura 3.3 ilustra la estructura de

entorno creada al evaluar la expresión `(square 5)` en el entorno global, donde `square` es el procedimiento generado en la Figura 3.2. Aplicar el procedimiento resulta en la creación de un nuevo entorno, etiquetado E1 en la figura, que comienza con un marco en el que `x`, el parámetro formal para el procedimiento, está ligado al argumento 5. El puntero que sube desde este marco muestra que el entorno envolvente del marco es el entorno global. El entorno global se elige aquí, porque este es el entorno que se indica como parte del objeto de procedimiento `square`. Dentro de E1, evaluamos el cuerpo del procedimiento, `(* x x)`. Dado que el valor de `x` en E1 es 5, el resultado es `(* 5 5)`, o 25.

Figura 3.3: Entorno creado al evaluar `(square 5)` en el entorno global.

El modelo de entorno de aplicación de procedimientos se puede resumir en dos reglas:

- Un objeto de procedimiento se aplica a un conjunto de argumentos construyendo un marco, ligando los parámetros formales del procedimiento a los argumentos de la llamada, y luego evaluando el cuerpo del procedimiento en el contexto del nuevo entorno construido. El nuevo marco tiene como entorno envolvente el entorno parte del objeto de procedimiento que se está aplicando.

- Un procedimiento se crea evaluando una expresión λ relativa a un entorno dado. El objeto de procedimiento resultante es un par que consiste en el texto de la expresión λ y un puntero al entorno en el que se creó el procedimiento.

También especificamos que definir un símbolo usando `define` crea una ligadura en el marco del entorno actual y asigna al símbolo el valor indicado.[13] Finalmente, especificamos el comportamiento de `set!`, la operación que nos obligó a introducir el modelo de entorno en primer lugar. Evaluar la expresión `(set!` ⟨*variable*$\kern0.08em\rangle$ ⟨*valor*$ \kern0.08em\rangle$`)` en algún entorno localiza la ligadura de la variable en el entorno y cambia esa ligadura para indicar el nuevo valor. Es decir, se encuentra el primer marco en el entorno que contiene una ligadura para la variable y se modifica ese marco. Si la variable no está ligada en el entorno, entonces `set!` señala un error.

Estas reglas de evaluación, aunque considerablemente más complejas que el modelo de sustitución, siguen siendo razonablemente sencillas. Además, el modelo de evaluación, aunque abstracto, proporciona una descripción correcta de cómo el intérprete evalúa las expresiones. En el Capítulo 4

veremos cómo este modelo puede servir como un plan para implementar un intérprete funcional. Las siguientes secciones elaboran los detalles del modelo analizando algunos programas ilustrativos.

Aplicando Procedimientos Simples

Cuando introdujimos el modelo de sustitución en Section 1.1.5 mostramos cómo la combinación `(f 5)` se evalúa a 136, dadas las siguientes definiciones de procedimientos:

(define (square x) (* x x)) (define (sum-of-squares x y) (+ (square x) (square y))) (define (f a) (sum-of-squares (+ a 1) (* a 2)))

Podemos analizar el mismo ejemplo usando el modelo de entorno. La Figura 3.4 muestra los tres objetos de procedimiento creados al evaluar las definiciones de `f`, `square`, y `sum/of/squares` en el entorno global. Cada objeto de procedimiento consiste en algún código, junto con un puntero al entorno global.

Figura 3.4: Objetos de procedimiento en el marco global.

En la Figura 3.5 vemos la estructura de entorno creada al evaluar la expresión `(f 5)`. La llamada a `f` crea un nuevo entorno E1 comenzando con un marco en el que `a`, el parámetro formal de `f`, está ligado al argumento 5. En E1, evaluamos el cuerpo de `f`:

(sum-of-squares (+ a 1) (* a 2))

Para evaluar esta combinación, primero evaluamos las subexpresiones. La primera subexpresión, `sum/of/squares`, tiene un valor que es un objeto de procedimiento. (Nótese cómo se encuentra este valor: primero miramos en el primer marco de E1, que no contiene ninguna ligadura para `sum/of/squares`. Luego procedemos al entorno envolvente, es decir, el entorno global, y encontramos la ligadura que se muestra en la Figura 3.4.) Las otras dos subexpresiones se evalúan aplicando las operaciones primitivas `+` y para evaluar las dos combinaciones `(+ a 1)` y `( a 2)` para obtener 6 y 10, respectivamente. Ahora aplicamos el objeto de procedimiento `sum/of/squares` a los argumentos 6 y 10. Esto resulta en un nuevo entorno E2 en el que los parámetros formales `x` y `y` están ligados a los argumentos. Dentro de E2 evaluamos la combinación `(+ (square x) (square y))`. Esto nos lleva a evaluar `(square x)`, donde `square` se encuentra en el marco global y `x` es 6. Una vez más, establecemos un nuevo entorno, E3, en el que `x` está ligado a 6,

y dentro de este evaluamos el cuerpo de `square`, que es `(* x x)`. También como parte de la aplicación de `sum/of/squares`, debemos evaluar la subexpresión `(square y)`, donde `y` es 10. Esta segunda llamada a `square` crea otro entorno, E4, en el que `x`, el parámetro formal de `square`, está ligado a 10. Y dentro de E4 debemos evaluar `(* x x)`.

Figura 3.5: Entornos creados al evaluar `(f 5)` usando los procedimientos de la Figura 3.4.

El punto importante a observar es que cada llamada a `square` crea un nuevo entorno que contiene una ligadura para `x`. Podemos ver aquí cómo los diferentes marcos sirven para mantener separadas las diferentes variables locales todas llamadas `x`. Nótese que cada marco creado por `square` apunta al entorno global, ya que este es el entorno indicado por el objeto de procedimiento `square`. Después de que se evalúan las subexpresiones, se devuelven los resultados. Los valores generados por las dos llamadas a `square` se suman por `sum/of/squares`, y este resultado es devuelto por `f`. Dado que nuestro enfoque aquí está en las estructuras de entorno, no nos detendremos en cómo estos valores devueltos se pasan de una llamada a otra; sin embargo, este también es un aspecto importante del proceso de evaluación, y volveremos a él en detalle en el Capítulo 5.

Ejercicio 3.9: En Section 1.2.1 usamos el modelo de sustitución para analizar dos procedimientos para calcular factoriales, una versión recursiva

(define (factorial n) (if (= n 1) 1 (* n (factorial (- n 1)))))

y una versión iterativa

(define (factorial n) (fact-iter 1 1 n)) (define (fact-iter product counter max-count) (if (> counter max-count) product (fact-iter (* counter product) (+ counter 1) max-count)))

Muestre las estructuras de entorno creadas al evaluar `(factorial 6)` usando cada versión del procedimiento `factorial`.[14]

Los Marcos como Repositorio del Estado Local

Podemos recurrir al modelo de entorno para ver cómo los procedimientos y la asignación pueden usarse para representar objetos con estado local. Como ejemplo, considere el "procesador de retiros" de Section 3.1.1 creado al

llamar al procedimiento

(define (make-withdraw balance) (lambda (amount) (if (>= balance amount) (begin (set! balance (- balance amount)) balance) "Fondos insuficientes")))

Describamos la evaluación de

(define W1 (make-withdraw 100))

seguido de

(W1 50) *50*

La Figura 3.6 muestra el resultado de definir el procedimiento `make/withdraw` en el entorno global. Esto produce un objeto de procedimiento que contiene un puntero al entorno global. Hasta ahora, esto no es diferente de los ejemplos que ya hemos visto, excepto que el cuerpo del procedimiento es en sí mismo una expresión λ.

Figura 3.6: Resultado de definir `make/withdraw` en el entorno global.

Figura 3.7: Resultado de evaluar `(define W1 (make/withdraw 100))`.

La parte interesante del cálculo ocurre cuando aplicamos el procedimiento `make/withdraw` a un argumento:

(define W1 (make-withdraw 100))

Comenzamos, como de costumbre, estableciendo un entorno E1 en el que el parámetro formal `balance` está ligado al argumento 100. Dentro de este entorno, evaluamos el cuerpo de `make/withdraw`, a saber, la expresión λ. Esto construye un nuevo objeto de procedimiento, cuyo código es como lo especifica la `lambda` y cuyo entorno es E1, el entorno en el que se evaluó la `lambda` para producir el procedimiento. El objeto de procedimiento resultante es el valor devuelto por la llamada a `make/withdraw`. Este está ligado a `W1` en el entorno global, ya que la propia `define` se está evaluando en el entorno global. La Figura 3.7 muestra la estructura de entorno resultante.

Figura 3.8: Entornos creados al aplicar el objeto de procedimiento `W1`.

Ahora podemos analizar lo que sucede cuando se aplica `W1` a un argumento:

(W1 50) *50*

Comenzamos construyendo un marco en el que `amount`, el parámetro formal de `W1`, está ligado al argumento 50. El punto crucial a observar es que este marco tiene como entorno envolvente no el entorno global, sino el entorno E1, porque este es el entorno que se especifica por el objeto de procedimiento `W1`. Dentro de este nuevo entorno, evaluamos el cuerpo del procedimiento:

(if (>= balance amount) (begin (set! balance (- balance amount)) balance) "Fondos insuficientes")

La estructura de entorno resultante se muestra en la Figura 3.8. La expresión que se está evaluando hace referencia tanto a `amount` como a `balance`. `amount` se encontrará en el primer marco del entorno, mientras que `balance` se encontrará siguiendo el puntero del entorno envolvente a E1.

Figura 3.9: Entornos después de la llamada a `W1`.

Cuando se ejecuta el `set!`, se cambia la ligadura de `balance` en E1. Al finalizar la llamada a `W1`, `balance` es 50, y el marco que contiene `balance` todavía es apuntado por el objeto de procedimiento `W1`. El marco que liga `amount` (en el que ejecutamos el código que cambió `balance`) ya no es relevante, ya que la llamada al procedimiento que lo construyó ha terminado, y no hay punteros a ese marco desde otras partes del entorno. La próxima vez que se llame a `W1`, esto construirá un nuevo marco que liga `amount` y cuyo entorno envolvente es E1. Vemos que E1 sirve como el "lugar" que contiene la variable de estado local para el objeto de procedimiento `W1`. La Figura 3.9 muestra la situación después de la llamada a `W1`. Observe lo que sucede cuando creamos un segundo objeto "retiro" haciendo otra llamada a `make/withdraw`:

(define W2 (make-withdraw 100))

Figura 3.10: Usando `(define W2 (make/withdraw 100))` para crear un segundo objeto.

Esto produce la estructura de entorno de la Figura 3.10, que muestra que `W2` es un objeto de procedimiento, es decir, un par con algún código y un entorno. El entorno E2 para `W2` fue creado por la llamada a `make/withdraw`.

Contiene un marco con su propia ligadura local para `balance`. Por otro lado, `W1` y `W2` tienen el mismo código: el código especificado por la expresión λ en el cuerpo de `make/withdraw`.[15] Vemos aquí por qué `W1` y `W2` se comportan como objetos independientes. Las llamadas a `W1` hacen referencia a la variable de estado `balance` almacenada en E1, mientras que las llamadas a `W2` hacen referencia al `balance` almacenado en E2. Por lo tanto, los cambios en el estado local de un objeto no afectan al otro objeto.

Ejercicio 3.10: En el procedimiento `make/withdraw`, la variable local `balance` se crea como un parámetro de `make/withdraw`. También podríamos crear la variable de estado local explícitamente, usando `let`, de la siguiente manera:

(define (make-withdraw initial-amount) (let ((balance initial-amount)) (lambda (amount) (if (>= balance amount) (begin (set! balance (- balance amount)) balance) "Fondos insuficientes"))))

Recuerde de Section 1.3.2 que `let` es simplemente azúcar sintáctico para una llamada a procedimiento:

(let ((⟨ *var* ⟩ ⟨ *exp* ⟩)) ⟨ *cuerpo* ⟩)

se interpreta como una sintaxis alternativa para

((lambda (⟨ *var* ⟩) ⟨ *cuerpo* ⟩) ⟨ *exp* ⟩)

Use el modelo de entorno para analizar esta versión alternativa de `make/withdraw`, dibujando figuras como las anteriores para ilustrar las interacciones

(define W1 (make-withdraw 100)) (W1 50) (define W2 (make-withdraw 100))

Muestre que las dos versiones de `make/withdraw` crean objetos con el mismo comportamiento. ¿En qué se diferencian las estructuras de entorno para las dos versiones?

Definiciones Internas

La Section 1.1.8 introdujo la idea de que los procedimientos pueden tener definiciones internas, lo que lleva a una estructura de bloques como en el siguiente procedimiento para calcular raíces cuadradas:

(define (sqrt x) (define (good-enough? guess) (< (abs (- (square guess) x)) 0.001)) (define (improve guess) (average guess (/ x guess))) (define (sqrt-iter

guess) (if (good-enough? guess) guess (sqrt-iter (improve guess)))) (sqrt-iter 1.0))

Ahora podemos usar el modelo de entorno para ver por qué estas definiciones internas se comportan como se desea. La Figura 3.11 muestra el punto en la evaluación de la expresión `(sqrt 2)` donde el procedimiento interno `good/enough?` ha sido llamado por primera vez con `guess` igual a 1. Observe la estructura del entorno. `sqrt` es un símbolo en el entorno global que está ligado a un objeto de procedimiento cuyo entorno asociado es el entorno global. Cuando se llamó a `sqrt`, se formó un nuevo entorno E1, subordinado al entorno global, en el que el parámetro `x` está ligado a 2. El cuerpo de `sqrt` se evaluó entonces en E1. Dado que la primera expresión en el cuerpo de `sqrt` es

(define (good-enough? guess) (< (abs (- (square guess) x)) 0.001))

evaluar esta expresión definió el procedimiento `good/enough?` en el entorno E1. Para ser más precisos, el símbolo `good/enough?` se agregó al primer marco de E1, ligado a un objeto de procedimiento cuyo entorno asociado es E1. De manera similar, `improve` y `sqrt/iter` se definieron como procedimientos en E1. Para mayor concisión, la Figura 3.11 muestra solo el objeto de procedimiento para `good/enough?`.

Figura 3.11: Procedimiento `sqrt` con definiciones internas.

Después de que se definieron los procedimientos locales, la expresión `(sqrt/iter 1.0)` se evaluó, todavía en el entorno E1. Así que el objeto de procedimiento ligado a `sqrt/iter` en E1 se llamó con 1 como argumento. Esto creó un entorno E2 en el que `guess`, el parámetro de `sqrt/iter`, está ligado a 1. `sqrt/iter` a su vez llamó a `good/enough?` con el valor de `guess` (de E2) como argumento para `good/enough?`. Esto estableció otro entorno, E3, en el que `guess` (el parámetro de `good/enough?`) está ligado a 1. Aunque `sqrt/iter` y `good/enough?` tienen ambos un parámetro llamado `guess`, estas son dos variables locales distintas ubicadas en marcos diferentes. Además, tanto E2 como E3 tienen E1 como su entorno envolvente, porque los procedimientos `sqrt/iter` y `good/enough?` tienen ambos E1 como su parte de entorno. Una consecuencia de esto es que el símbolo `x` que aparece en el cuerpo de `good/enough?` hará referencia a la ligadura de `x` que aparece en E1, es decir, el valor de `x` con el que se llamó al procedimiento `sqrt` original. El modelo de entorno explica así las dos propiedades clave que hacen que las definiciones de procedimientos locales sean una técnica útil para modularizar programas:

- Los nombres de los procedimientos locales no interfieren con los nombres externos al procedimiento envolvente, porque los nombres de los procedimientos locales estarán ligados en el marco que el procedimiento crea cuando se ejecuta, en lugar de estar ligados en el entorno global.

- Los procedimientos locales pueden acceder a los argumentos del procedimiento envolvente, simplemente usando los nombres de los parámetros como variables libres. Esto se debe a que el cuerpo del procedimiento local se evalúa en un entorno que está subordinado al entorno de evaluación del procedimiento envolvente.

  Ejercicio 3.11: En Section 3.2.3 vimos cómo el modelo de entorno describía el comportamiento de los procedimientos con estado local. Ahora hemos visto cómo funcionan las definiciones internas. Un procedimiento típico de paso de mensajes contiene ambos aspectos. Considere el procedimiento de cuenta bancaria de Section 3.1.1:

  (define (make-account balance) (define (withdraw amount) (if (>= balance amount) (begin (set! balance (- balance amount)) balance) "Fondos insuficientes")) (define (deposit amount) (set! balance (+ balance amount)) balance) (define (dispatch m) (cond ((eq? m 'withdraw) withdraw) ((eq? m 'deposit) deposit) (else (error "Solicitud desconocida: MAKE-ACCOUNT" m)))) dispatch)

  Muestre la estructura de entorno generada por la secuencia de interacciones

  (define acc (make-account 50)) ((acc 'deposit) 40) *90* ((acc 'withdraw) 60) *30*

  ¿Dónde se guarda el estado local de `acc`? Supongamos que definimos otra cuenta

  (define acc2 (make-account 100))

  ¿Cómo se mantienen distintos los estados locales de las dos cuentas? ¿Qué partes de la estructura del entorno se comparten entre `acc` y `acc2`?

Modelado con Datos Mutables

El Capítulo 2 trató sobre los datos compuestos como un medio para construir objetos computacionales que tienen varias partes, con el fin de modelar objetos del mundo real que tienen varios aspectos. En ese capítulo

introdujimos la disciplina de la abstracción de datos, según la cual las estructuras de datos se especifican en términos de constructores, que crean objetos de datos, y selectores, que acceden a las partes de los objetos de datos compuestos. Pero ahora sabemos que hay otro aspecto de los datos que el Capítulo 2 no abordó. El deseo de modelar sistemas compuestos por objetos que tienen un estado cambiante nos lleva a la necesidad de modificar los objetos de datos compuestos, así como de construirlos y seleccionarlos. Para modelar objetos compuestos con estado cambiante, diseñaremos abstracciones de datos para incluir, además de selectores y constructores, operaciones llamadas *mutadores*, que modifican los objetos de datos. Por ejemplo, modelar un sistema bancario requiere que cambiemos los saldos de las cuentas. Por lo tanto, una estructura de datos para representar cuentas bancarias podría admitir una operación

(set-balance! ⟨ *cuenta* ⟩ ⟨ *nuevo-valor* ⟩)

que cambia el saldo de la cuenta designada al nuevo valor designado. Los objetos de datos para los cuales se definen mutadores se conocen como *objetos de datos mutables*. El Capítulo 2 introdujo los pares como un "pegamento" de propósito general para sintetizar datos compuestos. Comenzamos esta sección definiendo mutadores básicos para los pares, de modo que los pares puedan servir como bloques de construcción para construir objetos de datos mutables. Estos mutadores aumentan enormemente el poder de representación de los pares, permitiéndonos construir estructuras de datos distintas de las secuencias y árboles con los que trabajamos en Section 2.2. También presentamos algunos ejemplos de simulaciones en las que sistemas complejos se modelan como colecciones de objetos con estado local.

Estructura de Lista Mutable

Las operaciones básicas sobre pares—`cons`, `car` y `cdr`—pueden usarse para construir estructuras de lista y para seleccionar partes de estructuras de lista, pero son incapaces de modificar estructuras de lista. Lo mismo es cierto para las operaciones de lista que hemos usado hasta ahora, como `append` y `list`, ya que estas pueden definirse en términos de `cons`, `car` y `cdr`. Para modificar estructuras de lista necesitamos nuevas operaciones. Los mutadores primitivos para los pares son `set/car!` y `set/cdr!`. `set/car!` toma dos argumentos, el primero de los cuales debe ser un par. Modifica este par, reemplazando el puntero `car` por un puntero al segundo argumento de `set/car!`.[16] Como ejemplo, supongamos que `x` está ligado a la lista `((a b) c d)` y `y` a la lista `(e f)` como se ilustra en la Figura 3.12. Evaluar la expresión `(set/car! x y)` modifica el par al que `x` está ligado, reemplazando su `car`

por el valor de `y`. El resultado de la operación se muestra en la Figura 3.13. La estructura `x` ha sido modificada y ahora se imprimiría como `((e f) c d)`. Los pares que representan la lista `(a b)`, identificados por el puntero que fue reemplazado, ahora están desvinculados de la estructura original.[17] Compare la Figura 3.13 con la Figura 3.14, que ilustra el resultado de ejecutar `(define z (cons y (cdr x)))` con `x` y `y` ligados a las listas originales de la Figura 3.12. La variable `z` ahora está ligada a un nuevo par creado por la operación `cons`; la lista a la que `x` está ligada no ha cambiado.

Figura 3.12: Listas `x`: `((a b) c d)` y `y`: `(e f)`.

Figura 3.13: Efecto de `(set/car! x y)` en las listas de la Figura 3.12.

Figura 3.14: Efecto de `(define z (cons y (cdr x)))` en las listas de la Figura 3.12.

Figura 3.15: Efecto de `(set/cdr! x y)` en las listas de la Figura 3.12.

La operación `set/cdr!` es similar a `set/car!`. La única diferencia es que el puntero `cdr` del par, en lugar del puntero `car`, es reemplazado. El efecto de ejecutar `(set/cdr! x y)` en las listas de la Figura 3.12 se muestra en la Figura 3.15. Aquí el puntero `cdr` de `x` ha sido reemplazado por el puntero a `(e f)`. Además, la lista `(c d)`, que solía ser el `cdr` de `x`, ahora está desvinculada de la estructura. `cons` construye una nueva estructura de lista creando nuevos pares, mientras que `set/car!` y `set/cdr!` modifican pares existentes. De hecho, podríamos implementar `cons` en términos de los dos mutadores, junto con un procedimiento `get/new/pair`, que devuelve un nuevo par que no es parte de ninguna estructura de lista existente. Obtenemos el nuevo par, establecemos sus punteros `car` y `cdr` a los objetos designados, y devolvemos el nuevo par como resultado del `cons`.[18]

(define (cons x y) (let ((new (get-new-pair))) (set-car! new x) (set-cdr! new y) new))

Ejercicio 3.12: El siguiente procedimiento para añadir listas fue introducido en Section 2.2.1:

(define (append x y) (if (null? x) y (cons (car x) (append (cdr x) y))))

`append` forma una nueva lista haciendo `cons` sucesivamente de los elementos de `x` en `y`. El procedimiento `append!` es similar a `append`, pero es un mutador en lugar de un constructor. Añade las listas empalmándolas, modificando el par final de `x` de modo que su `cdr` sea ahora `y`. (Es un error llamar a `append!` con un `x` vacío.)

(define (append! x y) (set-cdr! (last-pair x) y) x)

Aquí `last/pair` es un procedimiento que devuelve el último par de su argumento:

(define (last-pair x) (if (null? (cdr x)) x (last-pair (cdr x))))

Considere la interacción

(define x (list 'a 'b)) (define y (list 'c 'd)) (define z (append x y)) z *(a b c d)* (cdr x) ⟨ *respuesta* ⟩ (define w (append! x y)) w *(a b c d)* (cdr x) ⟨ *respuesta* ⟩

¿Cuáles son las ⟨*respuestas*⟩ faltantes? Dibuja diagramas de caja y puntero para explicar tu respuesta.

Ejercicio 3.13: Considere el siguiente procedimiento `make/cycle`, que utiliza el procedimiento `last/pair` definido en el Ejercicio 3.12:

(define (make-cycle x) (set-cdr! (last-pair x) x) x)

Dibuja un diagrama de caja y puntero que muestre la estructura `z` creada por

(define z (make-cycle (list 'a 'b 'c)))

¿Qué sucede si intentamos calcular `(last/pair z)`?

Ejercicio 3.14: El siguiente procedimiento es bastante útil, aunque oscuro:

(define (mystery x) (define (loop x y) (if (null? x) y (let ((temp (cdr x))) (set-cdr! x y) (loop temp x)))) (loop x '()))

`loop` usa la variable "temporal" `temp` para mantener el valor antiguo del `cdr` de `x`, ya que el `set/cdr!` en la siguiente línea destruye el `cdr`. Explica qué hace `mystery` en general. Supón que `v` se define por `(define v (list 'a 'b 'c 'd))`. Dibuja el diagrama de caja y puntero que

representa la lista a la que `v` está ligada. Supón que ahora evaluamos `(define w (mystery v))`. Dibuja diagramas de caja y puntero que muestren las estructuras `v` y `w` después de evaluar esta expresión. ¿Qué se imprimiría como los valores de `v` y `w`?

Compartición e identidad

Mencionamos en Section 3.1.3 los problemas teóricos de "identidad" y "cambio" planteados por la introducción de la asignación. Estos problemas surgen en la práctica cuando los pares individuales son *compartidos* entre diferentes objetos de datos. Por ejemplo, considere la estructura formada por

(define x (list ’a ’b)) (define z1 (cons x x))

Como se muestra en la Figura 3.16, `z1` es un par cuyo `car` y `cdr` apuntan ambos al mismo par `x`. Esta compartición de `x` por el `car` y `cdr` de `z1` es una consecuencia de la forma directa en que se implementa `cons`. En general, usar `cons` para construir listas resultará en una estructura interconectada de pares en la que muchos pares individuales son compartidos por muchas estructuras diferentes. En contraste con la Figura 3.16, la Figura 3.17 muestra la estructura creada por

(define z2 (cons (list ’a ’b) (list ’a ’b)))

En esta estructura, los pares en las dos listas `(a b)` son distintos, aunque los símbolos reales son compartidos.[19]

Figura 3.16: La lista `z1` formada por `(cons x x)`.

Figura 3.17: La lista `z2` formada por `(cons (list ’a ’b) (list ’a ’b))`.

Cuando se piensa en ellas como una lista, tanto `z1` como `z2` representan "la misma" lista, `((a b) a b)`. En general, la compartición es completamente indetectable si operamos sobre listas usando solo `cons`, `car` y `cdr`. Sin embargo, si permitimos mutadores en la estructura de la lista, la compartición se vuelve significativa. Como ejemplo de la diferencia que puede hacer la compartición, considere el siguiente procedimiento, que modifica el `car` de la estructura a la que se aplica:

(define (set-to-wow! x) (set-car! (car x) ’wow) x)

Aunque `z1` y `z2` son "la misma" estructura, aplicarles `set/to/wow!` produce resultados diferentes. Con `z1`, alterar el `car` también cambia el `cdr`, porque en `z1` el `car` y el `cdr` son el mismo par. Con `z2`, el `car` y el `cdr` son distintos, por lo que `set/to/wow!` modifica solo el `car`:

z1 *((a b) a b)* (set-to-wow! z1) *((wow b) wow b)* z2 *((a b) a b)* (set-to-wow! z2) *((wow b) a b)*

Una forma de detectar la compartición en estructuras de lista es usar el predicado `eq?`, que introdujimos en Section 2.3.1 como una forma de probar si dos símbolos son iguales. Más generalmente, `(eq? x y)` prueba si `x` y `y` son el mismo objeto (es decir, si `x` y `y` son iguales como punteros). Así, con `z1` y `z2` como se definieron en la Figura 3.16 y la Figura 3.17, `(eq? (car z1) (cdr z1))` es verdadero y `(eq? (car z2) (cdr z2))` es falso. Como se verá en las siguientes secciones, podemos explotar la compartición para extender en gran medida el repertorio de estructuras de datos que pueden ser representadas por pares. Por otro lado, la compartición también puede ser peligrosa, ya que las modificaciones hechas a las estructuras también afectarán a otras estructuras que casualmente comparten las partes modificadas. Las operaciones de mutación `set/car!` y `set/cdr!` deben usarse con cuidado; a menos que tengamos un buen entendimiento de cómo se comparten nuestros objetos de datos, la mutación puede tener resultados imprevistos.[20]

Ejercicio 3.15: Dibuja diagramas de caja y puntero para explicar el efecto de `set/to/wow!` en las estructuras `z1` y `z2` anteriores.

Ejercicio 3.16: Ben Bitdiddle decide escribir un procedimiento para contar el número de pares en cualquier estructura de lista. "Es fácil", razona. "El número de pares en cualquier estructura es el número en el `car` más el número en el `cdr` más uno más para contar el par actual." Así que Ben escribe el siguiente procedimiento:

```
(define (count-pairs x) (if (not (pair? x)) 0 (+ (count-pairs (car x)) (count-pairs (cdr x)) 1)))
```

Muestre que este procedimiento no es correcto. En particular, dibuje diagramas de caja y puntero que representen estructuras de lista compuestas exactamente por tres pares para las cuales el procedimiento de Ben devolvería 3; devolvería 4; devolvería 7; nunca devolvería nada.

Ejercicio 3.17: Diseñe una versión correcta del procedimiento `count/pairs` del Ejercicio 3.16 que devuelva el número de pares distintos en cualquier estructura. (Pista: Recorra la estructura, manteniendo una estructura de datos auxiliar que se utilice para llevar un registro de qué

pares ya se han contado.)

Ejercicio 3.18: Escriba un procedimiento que examine una lista y determine si contiene un ciclo, es decir, si un programa que intentara encontrar el final de la lista tomando sucesivos `cdr`s entraría en un bucle infinito. El Ejercicio 3.13 construyó tales listas.

Ejercicio 3.19: Rehaga el Ejercicio 3.18 usando un algoritmo que solo ocupe una cantidad constante de espacio. (Esto requiere una idea muy inteligente.)

La mutación es solo asignación

Cuando introdujimos los datos compuestos, observamos en Section 2.1.3 que los pares pueden representarse puramente en términos de procedimientos:

(define (cons x y) (define (dispatch m) (cond ((eq? m 'car) x) ((eq? m 'cdr) y) (else (error "Operación indefinida: CONS" m)))) dispatch) (define (car z) (z 'car)) (define (cdr z) (z 'cdr))

La misma observación es cierta para los datos mutables. Podemos implementar objetos de datos mutables como procedimientos usando asignación y estado local. Por ejemplo, podemos extender la implementación de pares anterior para manejar `set/car!` y `set/cdr!` de una manera análoga a como implementamos las cuentas bancarias usando `make/account` en Section 3.1.1:

(define (cons x y) (define (set-x! v) (set! x v)) (define (set-y! v) (set! y v)) (define (dispatch m) (cond ((eq? m 'car) x) ((eq? m 'cdr) y) ((eq? m 'set-car!) set-x!) ((eq? m 'set-cdr!) set-y!) (else (error "Operación indefinida: CONS" m)))) dispatch) (define (car z) (z 'car)) (define (cdr z) (z 'cdr)) (define (set-car! z new-value) ((z 'set-car!) new-value) z) (define (set-cdr! z new-value) ((z 'set-cdr!) new-value) z)

La asignación es todo lo que se necesita, teóricamente, para explicar el comportamiento de los datos mutables. Tan pronto como admitimos `set!` en nuestro lenguaje, planteamos todos los problemas, no solo de la asignación, sino de los datos mutables en general.[21]

Ejercicio 3.20: Dibuja diagramas de entorno para ilustrar la evaluación de la secuencia de expresiones

(define x (cons 1 2)) (define z (cons x x)) (set-car! (cdr z) 17) (car x) *17*

usando la implementación procedimental de pares dada anteriormente.

(Compare con el Ejercicio 3.11.)

Representando Colas

Los mutadores `set/car!` y `set/cdr!` nos permiten usar pares para construir estructuras de datos que no se pueden construir solo con `cons`, `car` y `cdr`. Esta sección muestra cómo usar pares para representar una estructura de datos llamada cola. El Section 3.3.3 mostrará cómo representar estructuras de datos llamadas tablas. Una *cola* es una secuencia en la que los elementos se insertan por un extremo (llamado la *parte trasera* de la cola) y se eliminan por el otro extremo (la *parte delantera*). La Figura 3.18 muestra una cola inicialmente vacía en la que se insertan los elementos `a` y `b`. Luego se elimina `a`, se insertan `c` y `d`, y se elimina `b`. Debido a que los elementos siempre se eliminan en el orden en que se insertan, una cola a veces se llama un búfer *FIFO* (primero en entrar, primero en salir).

Figura 3.18: Operaciones de cola.

En términos de abstracción de datos, podemos considerar una cola como definida por el siguiente conjunto de operaciones:

- un constructor: `(make/queue)` devuelve una cola vacía (una cola que no contiene elementos).
- dos selectores: `(empty/queue?` ⟨*cola*⟩`)` prueba si la cola está vacía. `(front/queue` ⟨*cola*⟩`)` devuelve el objeto al frente de la cola, señalando un error si la cola está vacía; no modifica la cola.
- dos mutadores: `(insert/queue!` ⟨*cola*⟩ ⟨*elemento*⟩`)` inserta el elemento al final de la cola y devuelve la cola modificada como su valor. `(delete/queue!` ⟨*cola*⟩`)` elimina el elemento al frente de la cola y devuelve la cola modificada como su valor, señalando un error si la cola está vacía antes de la eliminación.

Debido a que una cola es una secuencia de elementos, ciertamente podríamos representarla como una lista ordinaria; el frente de la cola sería el `car` de la lista, insertar un elemento en la cola equivaldría a añadir un nuevo elemento al final de la lista, y eliminar un elemento de la cola sería simplemente tomar el `cdr` de la lista. Sin embargo, esta representación es ineficiente, porque para insertar un elemento debemos recorrer la lista hasta llegar al final. Dado que el único método que tenemos para recorrer una lista es mediante operaciones sucesivas de `cdr`, este recorrido requiere $\Theta(n)$ pasos para una

lista de *n* elementos. Una simple modificación a la representación de lista supera esta desventaja al permitir que las operaciones de cola se implementen de modo que requieran $\Theta(1)$ pasos; es decir, de modo que el número de pasos necesarios sea independiente de la longitud de la cola. La dificultad con la representación de lista surge de la necesidad de escanear para encontrar el final de la lista. La razón por la que necesitamos escanear es que, aunque la forma estándar de representar una lista como una cadena de pares nos proporciona fácilmente un puntero al principio de la lista, no nos da un puntero fácilmente accesible al final. La modificación que evita el inconveniente es representar la cola como una lista, junto con un puntero adicional que indica el par final en la lista. De esa manera, cuando vayamos a insertar un elemento, podemos consultar el puntero trasero y así evitar escanear la lista. Una cola se representa, entonces, como un par de punteros, `front/ptr` y `rear/ptr`, que indican, respectivamente, el primer y el último par en una lista ordinaria. Como nos gustaría que la cola fuera un objeto identificable, podemos usar `cons` para combinar los dos punteros. Por lo tanto, la cola misma será el `cons` de los dos punteros. La Figura 3.19 ilustra esta representación.

Figura 3.19: Implementación de una cola como una lista con punteros frontal y trasero.

Para definir las operaciones de la cola usamos los siguientes procedimientos, que nos permiten seleccionar y modificar los punteros frontal y trasero de una cola:

(define (front-ptr queue) (car queue)) (define (rear-ptr queue) (cdr queue)) (define (set-front-ptr! queue item) (set-car! queue item)) (define (set-rear-ptr! queue item) (set-cdr! queue item))

Ahora podemos implementar las operaciones reales de la cola. Consideraremos que una cola está vacía si su puntero frontal es la lista vacía:

(define (empty-queue? queue) (null? (front-ptr queue)))

El constructor `make/queue` devuelve, como una cola inicialmente vacía, un par cuyo `car` y `cdr` son ambos la lista vacía:

(define (make-queue) (cons '() '()))

Para seleccionar el elemento al frente de la cola, devolvemos el `car` del par indicado por el puntero frontal:

(define (front-queue queue) (if (empty-queue? queue) (error "FRONT llamado con una cola vacía" queue) (car (front-ptr queue))))

Figura 3.20: Resultado de usar `(insert/queue! q 'd)` en la cola de la Figura 3.19.

Para insertar un elemento en una cola, seguimos el método cuyo resultado se indica en la Figura 3.20. Primero creamos un nuevo par cuyo `car` es el elemento a insertar y cuyo `cdr` es la lista vacía. Si la cola estaba inicialmente vacía, establecemos los punteros frontal y trasero de la cola a este nuevo par. De lo contrario, modificamos el par final en la cola para que apunte al nuevo par, y también establecemos el puntero trasero al nuevo par.

(define (insert-queue! queue item) (let ((new-pair (cons item '()))) (cond ((empty-queue? queue) (set-front-ptr! queue new-pair) (set-rear-ptr! queue new-pair) queue) (else (set-cdr! (rear-ptr queue) new-pair) (set-rear-ptr! queue new-pair) queue))))

Figura 3.21: Resultado de usar `(delete/queue! q)` en la cola de la Figura 3.20.

Para eliminar el elemento al frente de la cola, simplemente modificamos el puntero frontal para que ahora apunte al segundo elemento de la cola, que se puede encontrar siguiendo el puntero `cdr` del primer elemento (ver Figura 3.21):[22]

(define (delete-queue! queue) (cond ((empty-queue? queue) (error "DELETE! llamado con una cola vacía" queue)) (else (set-front-ptr! queue (cdr (front-ptr queue))) queue)))

Ejercicio 3.21: Ben Bitdiddle decide probar la implementación de la cola descrita anteriormente. Escribe los procedimientos en el intérprete de Lisp y procede a probarlos:

(define q1 (make-queue)) (insert-queue! q1 'a) *((a) a)* (insert-queue! q1 'b) *((a b) b)* (delete-queue! q1) *((b) b)* (delete-queue! q1) *(() b)*

"¡Está todo mal!", se queja. "La respuesta del intérprete muestra que el último elemento se inserta en la cola dos veces. Y cuando elimino ambos elementos, el segundo `b` sigue ahí, así que la cola no está vacía, aunque se supone que debería estarlo". Eva Lu Ator sugiere que Ben ha entendido

mal lo que está pasando. "No es que los elementos entren en la cola dos veces", explica. "Es solo que la impresora estándar de Lisp no sabe cómo dar sentido a la representación de la cola. Si quieres ver la cola impresa correctamente, tendrás que definir tu propio procedimiento de impresión para las colas". Explica de qué está hablando Eva Lu. En particular, muestra por qué los ejemplos de Ben producen los resultados impresos que producen. Define un procedimiento `print/queue` que tome una cola como entrada e imprima la secuencia de elementos en la cola.

Ejercicio 3.22: En lugar de representar una cola como un par de punteros, podemos construir una cola como un procedimiento con estado local. El estado local consistirá en punteros al principio y al final de una lista ordinaria. Así, el procedimiento `make/queue` tendrá la forma

(define (make-queue) (let ((front-ptr …) (rear-ptr …)) < *definiciones de procedimientos internos* > (define (dispatch m) …) dispatch))

Completa la definición de `make/queue` y proporciona implementaciones de las operaciones de la cola usando esta representación.

Ejercicio 3.23: Un *deque* ("cola de dos extremos") es una secuencia en la que los elementos pueden insertarse y eliminarse por el frente o por la parte trasera. Las operaciones en los deques son el constructor `make/deque`, el predicado `empty/deque?`, los selectores `front/deque` y `rear/deque`, los mutadores `front/insert/deque!`, `rear/insert/deque!`, `front/delete/deque!`, y `rear/delete/deque!`. Muestra cómo representar los deques usando pares, y da implementaciones de las operaciones.[23] Todas las operaciones deben realizarse en $\Theta(1)$ pasos.

Representando Tablas

Cuando estudiamos varias formas de representar conjuntos en el Capítulo 2, mencionamos en Section 2.3.3 la tarea de mantener una tabla de registros indexados por claves de identificación. En la implementación de la programación dirigida por datos en Section 2.4.3, hicimos un uso extensivo de tablas bidimensionales, en las que la información se almacena y recupera usando dos claves. Aquí vemos cómo construir tablas como estructuras de lista mutables.

Figura 3.22: Una tabla representada como una lista con encabezado.

Primero consideramos una tabla unidimensional, en la que cada valor se

almacena bajo una única clave. Implementamos la tabla como una lista de registros, cada uno de los cuales se implementa como un par que consiste en una clave y el valor asociado. Los registros se unen para formar una lista mediante pares cuyos `car` apuntan a registros sucesivos. Estos pares de unión se llaman el *esqueleto* de la tabla. Para tener un lugar que podamos cambiar cuando agreguemos un nuevo registro a la tabla, construimos la tabla como una *lista con encabezado*. Una lista con encabezado tiene un par de esqueleto especial al principio, que contiene un "registro" ficticio—en este caso, el símbolo elegido arbitrariamente `*table*`. La Figura 3.22 muestra el diagrama de caja y puntero para la tabla

a: 1 b: 2 c: 3

Para extraer información de una tabla usamos el procedimiento `lookup`, que toma una clave como argumento y devuelve el valor asociado (o falso si no hay ningún valor almacenado bajo esa clave). `lookup` se define en términos de la operación `assoc`, que espera una clave y una lista de registros como argumentos. Nótese que `assoc` nunca ve el registro ficticio. `assoc` devuelve el registro que tiene la clave dada como su `car`.[24] `lookup` luego comprueba si el registro resultante devuelto por `assoc` no es falso, y devuelve el valor (el `cdr`) del registro.

(define (lookup key table) (let ((record (assoc key (cdr table)))) (if record (cdr record) false))) (define (assoc key records) (cond ((null? records) false) ((equal? key (caar records)) (car records)) (else (assoc key (cdr records)))))

Para insertar un valor en una tabla bajo una clave especificada, primero usamos `assoc` para ver si ya hay un registro en la tabla con esta clave. Si no, formamos un nuevo registro haciendo `cons` de la clave con el valor, e insertamos esto en la cabecera de la lista de registros de la tabla, después del registro ficticio. Si ya hay un registro con esta clave, establecemos el `cdr` de este registro al nuevo valor designado. El encabezado de la tabla nos proporciona una ubicación fija para modificar con el fin de insertar el nuevo registro.[25]

(define (insert! key value table) (let ((record (assoc key (cdr table)))) (if record (set-cdr! record value) (set-cdr! table (cons (cons key value) (cdr table))))) 'ok)

Para construir una nueva tabla, simplemente creamos una lista que contiene el símbolo `*table*`:

(define (make-table) (list '*table*))

Tablas bidimensionales

En una tabla bidimensional, cada valor está indexado por dos claves. Podemos construir tal tabla como una tabla unidimensional en la que cada clave identifica una subtabla. La Figura 3.23 muestra el diagrama de caja y puntero para la tabla

math: +: 43 letters: a: 97 -: 45 b: 98 *: 42

que tiene dos subtablas. (Las subtablas no necesitan un símbolo de encabezado especial, ya que la clave que identifica la subtabla cumple esta función.)

Cuando buscamos un elemento, usamos la primera clave para identificar la subtabla correcta. Luego usamos la segunda clave para identificar el registro dentro de la subtabla.

(define (lookup key-1 key-2 table) (let ((subtable (assoc key-1 (cdr table)))) (if subtable (let ((record (assoc key-2 (cdr subtable)))) (if record (cdr record) false)) false)))

Figura 3.23: Una tabla bidimensional.

Para insertar un nuevo elemento bajo un par de claves, usamos `assoc` para ver si hay una subtabla almacenada bajo la primera clave. Si no, construimos una nueva subtabla que contiene el único registro (`key/2`, `value`) y la insertamos en la tabla bajo la primera clave. Si ya existe una subtabla para la primera clave, insertamos el nuevo registro en esta subtabla, usando el método de inserción para tablas unidimensionales descrito anteriormente:

(define (insert! key-1 key-2 value table) (let ((subtable (assoc key-1 (cdr table)))) (if subtable (let ((record (assoc key-2 (cdr subtable)))) (if record (set-cdr! record value) (set-cdr! subtable (cons (cons key-2 value) (cdr subtable))))) (set-cdr! table (cons (list key-1 (cons key-2 value)) (cdr table))))) 'ok)

Creando tablas locales

Las operaciones `lookup` e `insert!` definidas anteriormente toman la tabla como un argumento. Esto nos permite usar programas que acceden a más de una tabla. Otra forma de lidiar con múltiples tablas es tener procedimientos `lookup` e `insert!` separados para cada tabla. Podemos hacer esto representando una tabla procedimentalmente, como un objeto que mantiene

una tabla interna como parte de su estado local. Cuando se le envía un mensaje apropiado, este "objeto tabla" proporciona el procedimiento con el cual operar sobre la tabla interna. Aquí hay un generador para tablas bidimensionales representadas de esta manera:

(define (make-table) (let ((local-table (list '*table*))) (define (lookup key-1 key-2) (let ((subtable (assoc key-1 (cdr local-table)))) (if subtable (let ((record (assoc key-2 (cdr subtable)))) (if record (cdr record) false)) false))) (define (insert! key-1 key-2 value) (let ((subtable (assoc key-1 (cdr local-table)))) (if subtable (let ((record (assoc key-2 (cdr subtable)))) (if record (set-cdr! record value) (set-cdr! subtable (cons (cons key-2 value) (cdr subtable))))) (set-cdr! local-table (cons (list key-1 (cons key-2 value)) (cdr local-table))))) 'ok) (define (dispatch m) (cond ((eq? m 'lookup-proc) lookup) ((eq? m 'insert-proc!) insert!) (else (error "Unknown operation: TABLE" m)))) dispatch))

Usando `make/table`, podríamos implementar las operaciones `get` y `put` usadas en Section 2.4.3 para la programación dirigida por datos, de la siguiente manera:

(define operation-table (make-table)) (define get (operation-table 'lookup-proc)) (define put (operation-table 'insert-proc!))

`get` toma como argumentos dos claves, y `put` toma como argumentos dos claves y un valor. Ambas operaciones acceden a la misma tabla local, que está encapsulada dentro del objeto creado por la llamada a `make/table`.

Ejercicio 3.24: En las implementaciones de tablas anteriores, las claves se prueban para determinar su igualdad usando `equal?` (llamado por `assoc`). Esta no siempre es la prueba apropiada. Por ejemplo, podríamos tener una tabla con claves numéricas en la que no necesitamos una coincidencia exacta con el número que estamos buscando, sino solo un número dentro de cierta tolerancia. Diseñe un constructor de tabla `make/table` que tome como argumento un procedimiento `same/key?` que se usará para probar la "igualdad" de las claves. `make/table` debería devolver un procedimiento `dispatch` que pueda ser usado para acceder a los procedimientos `lookup` e `insert!` apropiados para una tabla local.

Ejercicio 3.25: Generalizando las tablas de una y dos dimensiones, muestre cómo implementar una tabla en la que los valores se almacenan bajo un número arbitrario de claves y diferentes valores pueden almacenarse bajo diferentes números de claves. Los procedimientos `lookup` e `insert!` deberían tomar como entrada una lista de claves utilizadas para acceder a la tabla.

Ejercicio 3.26: Para buscar en una tabla como la implementada anteriormente, es necesario escanear la lista de registros. Esta es básicamente la representación de lista desordenada de Section 2.3.3. Para tablas grandes, puede ser más eficiente estructurar la tabla de una manera diferente. Describa una implementación de tabla donde los registros (clave, valor) se organicen usando un árbol binario, asumiendo que las claves pueden ordenarse de alguna manera (por ejemplo, numérica o alfabéticamente). (Compare con el Ejercicio 2.66 del Capítulo 2.)

Ejercicio 3.27: La *memoización* (también llamada *tabulación*) es una técnica que permite a un procedimiento registrar, en una tabla local, valores que se han calculado previamente. Esta técnica puede marcar una gran diferencia en el rendimiento de un programa. Un procedimiento memoizado mantiene una tabla en la que los valores de las llamadas anteriores se almacenan utilizando como claves los argumentos que produjeron los valores. Cuando se le pide al procedimiento memoizado que calcule un valor, primero comprueba la tabla para ver si el valor ya está allí y, si es así, simplemente devuelve ese valor. De lo contrario, calcula el nuevo valor de la manera ordinaria y lo almacena en la tabla. Como ejemplo de memoización, recuerde de Section 1.2.2 el proceso exponencial para calcular los números de Fibonacci:

(define (fib n) (cond ((= n 0) 0) ((= n 1) 1) (else (+ (fib (- n 1)) (fib (- n 2))))))

La versión memoizada del mismo procedimiento es

(define memo-fib (memoize (lambda (n) (cond ((= n 0) 0) ((= n 1) 1) (else (+ (memo-fib (- n 1)) (memo-fib (- n 2)))))))

donde el memoizador se define como

(define (memoize f) (let ((table (make-table))) (lambda (x) (let ((previously-computed-result (lookup x table))) (or previously-computed-result (let ((result (f x))) (insert! x result table) result))))))

Dibuja un diagrama de entorno para analizar el cálculo de `(memo/fib 3)`. Explica por qué `memo/fib` calcula el número de Fibonacci n^{th} en un número de pasos proporcional a n. ¿Funcionaría el esquema si hubiéramos definido simplemente `memo/fib` como `(memoize fib)`?

Un Simulador para Circuitos Digitales

El diseño de sistemas digitales complejos, como las computadoras, es una importante actividad de ingeniería. Los sistemas digitales se construyen interconectando elementos simples. Aunque el comportamiento de estos elementos individuales es simple, las redes de ellos pueden tener un comportamiento muy complejo. La simulación por computadora de diseños de circuitos propuestos es una herramienta importante utilizada por los ingenieros de sistemas digitales. En esta sección diseñamos un sistema para realizar simulaciones de lógica digital. Este sistema tipifica un tipo de programa llamado *simulación dirigida por eventos*, en la que las acciones ("eventos") desencadenan más eventos que ocurren en un momento posterior, los que a su vez desencadenan más eventos, y así sucesivamente.

Nuestro modelo computacional de un circuito estará compuesto por objetos que corresponden a los componentes elementales a partir de los cuales se construye el circuito. Hay *cables*, que transportan *señales digitales*. Una señal digital puede en cualquier momento tener solo uno de dos valores posibles, 0 y 1. También hay varios tipos de *cajas de función* digitales, que conectan cables que transportan señales de entrada a otros cables de salida. Tales cajas producen señales de salida calculadas a partir de sus señales de entrada. La señal de salida se retrasa por un tiempo que depende del tipo de la caja de función. Por ejemplo, un *inversor* es una caja de función primitiva que invierte su entrada. Si la señal de entrada a un inversor cambia a 0, entonces un retardo de inversor después el inversor cambiará su señal de salida a 1. Si la señal de entrada a un inversor cambia a 1, entonces un retardo de inversor después el inversor cambiará su señal de salida a 0. Dibujamos un inversor simbólicamente como en la Figura 3.24. Una *compuerta AND*, también mostrada en la Figura 3.24, es una caja de función primitiva con dos entradas y una salida. Conduce su señal de salida a un valor que es el *AND lógico* de las entradas. Es decir, si ambas señales de entrada se vuelven 1, entonces un tiempo de retardo de compuerta AND después la compuerta AND forzará su señal de salida a ser 1; de lo contrario, la salida será 0. Una *compuerta OR* es una caja de función primitiva de dos entradas similar que conduce su señal de salida a un valor que es el *OR lógico* de las entradas. Es decir, la salida se volverá 1 si al menos una de las señales de entrada es 1; de lo contrario, la salida se volverá 0.

Figura 3.24: Funciones primitivas en el simulador de lógica digital.

Podemos conectar funciones primitivas entre sí para construir funciones más complejas. Para lograr esto, conectamos las salidas de algunas cajas de función a las entradas de otras cajas de función. Por ejemplo, el circuito *medio-sumador* mostrado en la Figura 3.25 consiste en una compuerta OR,

dos compuertas AND y un inversor. Toma dos señales de entrada, A y B, y tiene dos señales de salida, S y C. S se volverá 1 siempre que precisamente una de A y B sea 1, y C se volverá 1 siempre que A y B sean ambas 1. Podemos ver en la figura que, debido a los retardos involucrados, las salidas pueden generarse en diferentes momentos. Muchas de las dificultades en el diseño de circuitos digitales surgen de este hecho.

Figura 3.25: Un circuito de medio sumador.

Ahora construiremos un programa para modelar los circuitos de lógica digital que deseamos estudiar. El programa construirá objetos computacionales que modelan los cables, que "mantendrán" las señales. Las cajas de función serán modeladas por procedimientos que imponen las relaciones correctas entre las señales.

Un elemento básico de nuestra simulación será un procedimiento `make/wire`, que construye cables. Por ejemplo, podemos construir seis cables de la siguiente manera:

(define a (make-wire)) (define b (make-wire)) (define c (make-wire)) (define d (make-wire)) (define e (make-wire)) (define s (make-wire))

Conectamos una caja de función a un conjunto de cables llamando a un procedimiento que construye ese tipo de caja. Los argumentos del procedimiento constructor son los cables que se conectarán a la caja. Por ejemplo, dado que podemos construir compuertas AND, compuertas OR e inversores, podemos cablear el medio sumador que se muestra en la Figura 3.25:

(or-gate a b d) *ok* (and-gate a b c) *ok* (inverter c e) *ok* (and-gate d e s) *ok*

Mejor aún, podemos nombrar explícitamente esta operación definiendo un procedimiento `half/adder` que construye este circuito, dados los cuatro cables externos que se conectarán al medio sumador:

(define (half-adder a b s c) (let ((d (make-wire)) (e (make-wire))) (or-gate a b d) (and-gate a b c) (inverter c e) (and-gate d e s) 'ok))

La ventaja de hacer esta definición es que podemos usar `half/adder` mismo como un bloque de construcción para crear circuitos más complejos. La Figura 3.26, por ejemplo, muestra un *sumador completo* compuesto por dos medios sumadores y una compuerta OR.[26] Podemos construir un sumador completo de la siguiente manera:

```
(define (full-adder a b c-in sum c-out) (let ((s (make-wire)) (c1 (make-wire)) (c2 (make-wire))) (half-adder b c-in s c1) (half-adder a s sum c2) (or-gate c1 c2 c-out) 'ok))
```

Figura 3.26: Un circuito sumador completo.

Habiendo definido `full/adder` como un procedimiento, ahora podemos usarlo como un bloque de construcción para crear circuitos aún más complejos. (Por ejemplo, vea el Ejercicio 3.30.)

En esencia, nuestro simulador nos proporciona las herramientas para construir un lenguaje de circuitos. Si adoptamos la perspectiva general sobre los lenguajes con la que abordamos el estudio de Lisp en el Section 1.1, podemos decir que las cajas de función primitivas forman los elementos primitivos del lenguaje, que conectar cajas proporciona un medio de combinación, y que especificar patrones de conexión como procedimientos sirve como un medio de abstracción.

Cajas de función primitivas

Las cajas de función primitivas implementan las "fuerzas" por las cuales un cambio en la señal en un cable influye en las señales en otros cables. Para construir cajas de función, utilizamos las siguientes operaciones sobre los cables:

- `(get/signal` $\;\;\langle\kern0.06em\hbox{\ttfamily\slshape wire}\kern0.08em\rangle$`)`

  devuelve el valor actual de la señal en el cable.

- `(set/signal!` $\;\;\langle\kern0.08em\hbox{\ttfamily\slshape wire}\kern0.08em\rangle\;\;\langle\kern0.08em\hbox{\ttfamily \slshape new value}\kern0.08em\rangle$`)`

  cambia el valor de la señal en el cable al nuevo valor.

- `(add/action!` $\;\;\langle\kern0.08em\hbox{\ttfamily\slshape wire}\kern0.08em\rangle\;\;\langle\kern0.08em\hbox{\ttfamily \slshape procedure of no arguments}\kern0.02em\rangle$`)`

  afirma que el procedimiento designado debe ejecutarse cada vez que la señal en el cable cambie de valor. Tales procedimientos son los vehículos por los cuales los cambios en el valor de la señal en el cable se comunican

a otros cables.

Además, haremos uso de un procedimiento `after/delay` que toma un retraso de tiempo y un procedimiento a ejecutar y ejecuta el procedimiento dado después del retraso dado.

Usando estos procedimientos, podemos definir las funciones lógicas digitales primitivas. Para conectar una entrada a una salida a través de un inversor, usamos `add/action!` para asociar con el cable de entrada un procedimiento que se ejecutará cada vez que la señal en el cable de entrada cambie de valor. El procedimiento calcula la `negación/lógica` de la señal de entrada, y luego, después de un `retraso/de/inversor`, establece la señal de salida en este nuevo valor:

(define (inverter input output) (define (invert-input) (let ((new-value (logical-not (get-signal input)))) (after-delay inverter-delay (lambda () (set-signal! output new-value))))) (add-action! input invert-input) 'ok) (define (logical-not s) (cond ((= s 0) 1) ((= s 1) 0) (else (error "Invalid signal" s))))

Una compuerta Y es un poco más compleja. El procedimiento de acción debe ejecutarse si cualquiera de las entradas de la compuerta cambia. Calcula el `Y/lógico` (usando un procedimiento análogo a `no/lógico`) de los valores de las señales en los cables de entrada y establece un cambio al nuevo valor que ocurrirá en el cable de salida después de un `retraso/de/compuerta/Y`.

(define (and-gate a1 a2 output) (define (and-action-procedure) (let ((new-value (logical-and (get-signal a1) (get-signal a2)))) (after-delay and-gate-delay (lambda () (set-signal! output new-value))))) (add-action! a1 and-action-procedure) (add-action! a2 and-action-procedure) 'ok)

Ejercicio 3.28: Define una compuerta O como una caja de función primitiva. Tu constructor `or/gate` debería ser similar a `and/gate`.

Ejercicio 3.29: Otra forma de construir una compuerta O es como un dispositivo de lógica digital compuesto, construido a partir de compuertas Y e inversores. Define un procedimiento `or/gate` que logre esto. ¿Cuál es el tiempo de retardo de la compuerta O en términos de `and/gate/delay` e `inverter/delay`?

Ejercicio 3.30: La Figura 3.27 muestra un *sumador de acarreo en cascada* formado al encadenar *n* sumadores completos. Esta es la forma más simple de sumador paralelo para sumar dos números binarios de *n* bits. Las entradas A_1, A_2, A_3, …, A_n y B_1, B_2, B_3, …, B_n son los dos números binarios a sumar (cada A_k y B_k es un 0 o un 1). El circuito genera S_1, S_2, S_3, …, S_n, los *n* bits de la suma, y *C*, el acarreo de la adición. Escribe un

procedimiento `ripple/carry/adder` que genere este circuito. El procedimiento debería tomar como argumentos tres listas de *n* cables cada una—los *Ak*, los *Bk*, y los *Sk*—y también otro cable *C*. La principal desventaja del sumador de acarreo en cascada es la necesidad de esperar a que se propaguen las señales de acarreo. ¿Cuál es el retardo necesario para obtener la salida completa de un sumador de acarreo en cascada de *n* bits, expresado en términos de los retardos para las compuertas Y, las compuertas O y los inversores?

Figura 3.27: Un sumador de acarreo en cascada para números de *n* bits.

Representando cables

Un cable en nuestra simulación será un objeto computacional con dos variables de estado locales: un `signal/value` (inicialmente tomado como 0) y una colección de `action/procedures` que se ejecutarán cuando la señal cambie de valor. Implementamos el cable, usando un estilo de paso de mensajes, como una colección de procedimientos locales junto con un procedimiento `dispatch` que selecciona la operación local apropiada, tal como lo hicimos con el objeto simple de cuenta bancaria en Section 3.1.1:

(define (make-wire) (let ((signal-value 0) (action-procedures '())) (define (set-my-signal! new-value) (if (not (= signal-value new-value)) (begin (set! signal-value new-value) (call-each action-procedures)) 'done)) (define (accept-action-procedure! proc) (set! action-procedures (cons proc action-procedures)) (proc)) (define (dispatch m) (cond ((eq? m 'get-signal) signal-value) ((eq? m 'set-signal!) set-my-signal!) ((eq? m 'add-action!) accept-action-procedure!) (else (error "Unknown operation: WIRE" m)))) dispatch))

El procedimiento local `set/my/signal!` prueba si el nuevo valor de la señal cambia la señal en el cable. Si es así, ejecuta cada uno de los procedimientos de acción, usando el siguiente procedimiento `call/each`, que llama a cada uno de los elementos de una lista de procedimientos sin argumentos:

(define (call-each procedures) (if (null? procedures) 'done (begin ((car procedures)) (call-each (cdr procedures)))))

El procedimiento local `accept/action/procedure!` agrega el procedimiento dado a la lista de procedimientos a ejecutar, y luego ejecuta el nuevo procedimiento una vez. (Ver Ejercicio 3.31.)

Con el procedimiento local `dispatch` configurado como se especifica,

podemos proporcionar los siguientes procedimientos para acceder a las operaciones locales en los cables:[27]

(define (get-signal wire) (wire 'get-signal)) (define (set-signal! wire new-value) ((wire 'set-signal!) new-value)) (define (add-action! wire action-procedure) ((wire 'add-action!) action-procedure))

Los cables, que tienen señales que varían en el tiempo y pueden conectarse incrementalmente a dispositivos, son típicos de los objetos mutables. Los hemos modelado como procedimientos con variables de estado locales que se modifican por asignación. Cuando se crea un nuevo cable, se asigna un nuevo conjunto de variables de estado (mediante la expresión `let` en `make/wire`) y se construye y devuelve un nuevo procedimiento `dispatch`, capturando el entorno con las nuevas variables de estado.

Los cables se comparten entre los diversos dispositivos que se han conectado a ellos. Por lo tanto, un cambio realizado por una interacción con un dispositivo afectará a todos los demás dispositivos conectados al cable. El cable comunica el cambio a sus vecinos llamando a los procedimientos de acción que se le proporcionaron cuando se establecieron las conexiones.

La agenda

Lo único que falta para completar el simulador es `after/delay`. La idea aquí es que mantenemos una estructura de datos, llamada *agenda*, que contiene una programación de cosas por hacer. Las siguientes operaciones están definidas para las agendas:

- `(make/agenda)` devuelve una nueva agenda vacía.
- `(empty/agenda?`$\;\;\langle\kern0.08em\hbox{\ttfamily\slshape agenda}\kern0.06em\rangle\hbox{\tt)}$ es verdadero si la agenda especificada está vacía.
- `(first/agenda/item`$\;\;\langle\kern0.08em\hbox{\ttfamily\slshape agenda}\kern0.06em\rangle\hbox{\tt)}$ devuelve el primer elemento de la agenda.
- `(remove/first/agenda/item!`$\;\langle\kern0.08em\hbox{\ttfamily \slshape agenda}\kern0.06em\rangle\hbox{\tt)}$ modifica la agenda eliminando el primer elemento.
- `(add/to/agenda!`$\;\;\langle\kern0.03em\hbox{\ttfamily\slshape tiempo}\kern0.06em\rangle\;\;\langle\kern0.08em\hbox{\ttfamily \slshape acción}\kern0.06em\rangle\;\;\langle\kern0.08em

\hbox{\ttfamily\slshape agenda}\kern0.06em\rangle\hbox{\tt)}$ modifica la agenda añadiendo el procedimiento de acción dado para que se ejecute en el tiempo especificado.

- `(current/time`$\;\;\langle\kern0.08em\hbox{\ttfamily\slshape agenda}\kern0.04em\rangle\hbox{\tt)}$ devuelve el tiempo de simulación actual.

La agenda particular que usamos se denota por `the/agenda`. El procedimiento `after/delay` añade nuevos elementos a `the/agenda`:

(define (after-delay delay action) (add-to-agenda! (+ delay (current-time the-agenda)) action the-agenda))

La simulación es impulsada por el procedimiento `propagate`, que opera sobre `the/agenda`, ejecutando cada procedimiento en la agenda en secuencia. En general, a medida que la simulación se ejecuta, se añadirán nuevos elementos a la agenda, y `propagate` continuará la simulación mientras haya elementos en la agenda:

(define (propagate) (if (empty-agenda? the-agenda) 'done (let ((first-item (first-agenda-item the-agenda))) (first-item) (remove-first-agenda-item! the-agenda) (propagate))))

Una simulación de ejemplo

El siguiente procedimiento, que coloca una "sonda" en un cable, muestra el simulador en acción. La sonda le dice al cable que, cada vez que su señal cambie de valor, debe imprimir el nuevo valor de la señal, junto con el tiempo actual y un nombre que identifique el cable:

(define (probe name wire) (add-action! wire (lambda () (newline) (display name) (display " ") (display (current-time the-agenda)) (display " New-value = ") (display (get-signal wire)))))

Comenzamos inicializando la agenda y especificando los retardos para las cajas de función primitivas:

(define the-agenda (make-agenda)) (define inverter-delay 2) (define and-gate-delay 3) (define or-gate-delay 5)

Ahora definimos cuatro cables, colocando sondas en dos de ellos:

(define input-1 (make-wire)) (define input-2 (make-wire)) (define sum (make-wire)) (define carry (make-wire))

(probe 'sum sum) *sum 0 New-value = 0*

(probe 'carry carry) *carry 0 New-value = 0*

A continuación, conectamos los cables en un circuito de medio sumador (como en la Figura 3.25), establecemos la señal en `input/1` en 1 y ejecutamos la simulación:

(half-adder input-1 input-2 sum carry) *ok*

(set-signal! input-1 1) *done*

(propagate) *sum 8 New-value = 1 done*

La señal de `sum` cambia a 1 en el tiempo 8. Ahora estamos a ocho unidades de tiempo desde el comienzo de la simulación. En este punto, podemos establecer la señal en `input/2` en 1 y permitir que los valores se propaguen:

(set-signal! input-2 1) *done*

(propagate) *carry 11 New-value = 1 sum 16 New-value = 0 done*

La `carry` cambia a 1 en el tiempo 11 y la `sum` cambia a 0 en el tiempo 16.

> **Ejercicio 3.31:** El procedimiento interno `accept/action/procedure!` definido en `make/wire` especifica que cuando se agrega un nuevo procedimiento de acción a un cable, el procedimiento se ejecuta inmediatamente. Explique por qué es necesaria esta inicialización. En particular, siga el ejemplo del medio sumador en los párrafos anteriores y diga cómo diferiría la respuesta del sistema si hubiéramos definido `accept/action/procedure!` como
>
> (define (accept-action-procedure! proc) (set! action-procedures (cons proc action-procedures)))

Implementando la agenda

Finalmente, damos detalles de la estructura de datos de la agenda, que contiene los procedimientos que están programados para su ejecución futura.

La agenda está compuesta por *segmentos de tiempo*. Cada segmento de tiempo es un par que consiste en un número (el tiempo) y una cola (ver Ejercicio 3.32) que contiene los procedimientos que están programados para ejecutarse durante ese segmento de tiempo.

(define (make-time-segment time queue) (cons time queue)) (define (segment-time s) (car s)) (define (segment-queue s) (cdr s))

Operaremos sobre las colas de segmentos de tiempo usando las operaciones de cola descritas en Section 3.3.2.

La agenda en sí es una tabla unidimensional de segmentos de tiempo. Se diferencia de las tablas descritas en Section 3.3.3 en que los segmentos estarán ordenados en orden de tiempo creciente. Además, almacenamos el *tiempo actual* (es decir, el tiempo de la última acción que se procesó) al principio de la agenda. Una agenda recién construida no tiene segmentos de tiempo y tiene un tiempo actual de 0:[28]

(define (make-agenda) (list 0)) (define (current-time agenda) (car agenda)) (define (set-current-time! agenda time) (set-car! agenda time)) (define (segments agenda) (cdr agenda)) (define (set-segments! agenda segments) (set-cdr! agenda segments)) (define (first-segment agenda) (car (segments agenda))) (define (rest-segments agenda) (cdr (segments agenda)))

Una agenda está vacía si no tiene segmentos de tiempo:

(define (empty-agenda? agenda) (null? (segments agenda)))

Para agregar una acción a una agenda, primero verificamos si la agenda está vacía. Si es así, creamos un segmento de tiempo para la acción y lo instalamos en la agenda. De lo contrario, recorremos la agenda, examinando el tiempo de cada segmento. Si encontramos un segmento para nuestro tiempo designado, agregamos la acción a la cola asociada. Si llegamos a un tiempo posterior al que estamos designados, insertamos un nuevo segmento de tiempo en la agenda justo antes de él. Si llegamos al final de la agenda, debemos crear un nuevo segmento de tiempo al final.

(define (add-to-agenda! time action agenda) (define (belongs-before? segments) (or (null? segments) (< time (segment-time (car segments))))) (define (make-new-time-segment time action) (let ((q (make-queue))) (insert-queue! q action) (make-time-segment time q))) (define (add-to-segments! segments) (if (= (segment-time (car segments)) time) (insert-queue! (segment-queue (car segments)) action) (let ((rest (cdr segments))) (if (belongs-before? rest) (set-cdr! segments (cons (make-new-time-segment time action) (cdr segments))) (add-to-segments! rest))))) (let ((segments (segments agenda))) (if (belongs-before? segments) (set-segments! agenda (cons (make-new-time-segment time action) segments)) (add-to-segments! segments))))

El procedimiento que elimina el primer elemento de la agenda elimina el elemento al frente de la cola en el primer segmento de tiempo. Si esta

eliminación hace que el segmento de tiempo quede vacío, lo eliminamos de la lista de segmentos:[29]

(define (remove-first-agenda-item! agenda) (let ((q (segment-queue (first-segment agenda)))) (delete-queue! q) (if (empty-queue? q) (set-segments! agenda (rest-segments agenda)))))

El primer elemento de la agenda se encuentra al principio de la cola en el primer segmento de tiempo. Cada vez que extraemos un elemento, también actualizamos el tiempo actual:[30]

(define (first-agenda-item agenda) (if (empty-agenda? agenda) (error "Agenda is empty: FIRST-AGENDA-ITEM") (let ((first-seg (first-segment agenda))) (set-current-time! agenda (segment-time first-seg)) (front-queue (segment-queue first-seg)))))

> **Ejercicio 3.32:** Los procedimientos que se ejecutarán durante cada segmento de tiempo de la agenda se guardan en una cola. Por lo tanto, los procedimientos para cada segmento se llaman en el orden en que se agregaron a la agenda (primero en entrar, primero en salir). Explique por qué se debe usar este orden. En particular, siga el comportamiento de una compuerta Y cuyas entradas cambian de 0, 1 a 1, 0 en el mismo segmento y diga cómo diferiría el comportamiento si almacenáramos los procedimientos de un segmento en una lista ordinaria, agregando y eliminando procedimientos solo por el frente (último en entrar, primero en salir).

Propagación de Restricciones

Los programas de computadora se organizan tradicionalmente como cálculos unidireccionales, que realizan operaciones sobre argumentos preespecificados para producir salidas deseadas. Por otro lado, a menudo modelamos sistemas en términos de relaciones entre cantidades. Por ejemplo, un modelo matemático de una estructura mecánica podría incluir la información de que la deflexión d de una varilla de metal está relacionada con la fuerza F sobre la varilla, la longitud L de la varilla, el área de la sección transversal A, y el módulo elástico E a través de la ecuación

$$dAE = FL.$$

Tal ecuación no es unidireccional. Dadas cuatro de las cantidades, podemos usarla para calcular la quinta. Sin embargo, traducir la ecuación a un lenguaje de computadora tradicional nos obligaría a elegir una de las cantidades para ser calculada en términos de las otras cuatro. Así, un

procedimiento para calcular el área A no podría usarse para calcular la deflexión d, aunque los cálculos de A y d surgen de la misma ecuación.[31]

En esta sección, esbozamos el diseño de un lenguaje que nos permite trabajar en términos de las relaciones mismas. Los elementos primitivos del lenguaje son *restricciones primitivas*, que establecen que ciertas relaciones se mantienen entre cantidades. Por ejemplo, `(adder a b c)` especifica que las cantidades a, b y c deben estar relacionadas por la ecuación $a + b = c$, `(multiplier x y z)` expresa la restricción $xy = z$, y `(constant 3.14 x)` dice que el valor de x debe ser 3.14.

Nuestro lenguaje proporciona un medio para combinar restricciones primitivas con el fin de expresar relaciones más complejas. Combinamos restricciones construyendo *redes de restricciones*, en las que las restricciones se unen mediante *conectores*. Un conector es un objeto que "contiene" un valor que puede participar en una o más restricciones. Por ejemplo, sabemos que la relación entre las temperaturas Fahrenheit y Celsius es

$$9C = 5(F - 32).$$

Tal restricción puede considerarse como una red que consta de restricciones primitivas de sumador, multiplicador y constante (Figura 3.28). En la figura, vemos a la izquierda una caja de multiplicador con tres terminales, etiquetadas $m1$, $m2$, y p. Estas conectan el multiplicador al resto de la red de la siguiente manera: El terminal $m1$ está vinculado a un conector C, que contendrá la temperatura Celsius. El terminal $m2$ está vinculado a un conector w, que también está vinculado a una caja de constante que contiene 9. El terminal p, que la caja del multiplicador restringe a ser el producto de $m1$ y $m2$, está vinculado al terminal p de otra caja de multiplicador, cuyo $m2$ está conectado a una constante 5 y cuyo $m1$ está conectado a uno de los términos de una suma.

Figura 3.28: La relación $9C = 5(F - 32)$ expresada como una red de restricciones.

El cálculo mediante tal red procede de la siguiente manera: Cuando a un conector se le da un valor (por el usuario o por una caja de restricción a la que está vinculado), despierta a todas sus restricciones asociadas (excepto la restricción que acaba de despertarlo) para informarles que tiene un valor. Cada caja de restricción despertada luego sondea sus conectores para ver si hay suficiente información para determinar un valor para un conector. Si es así, la caja establece ese conector, que luego despierta a todas sus restricciones asociadas, y así sucesivamente. Por ejemplo, en la conversión

entre Celsius y Fahrenheit, *w*, *x*, y *y* son establecidos inmediatamente por las cajas de constantes a 9, 5 y 32, respectivamente. Los conectores despiertan a los multiplicadores y al sumador, que determinan que no hay suficiente información para proceder. Si el usuario (o alguna otra parte de la red) establece *C* en un valor (digamos 25), el multiplicador de la izquierda se despertará, y establecerá *u* en $25 \cdot 9 = 225$. Luego *u* despierta al segundo multiplicador, que establece *v* en 45, y *v* despierta al sumador, que establece *f* en 77.

Usando el sistema de restricciones

Para usar el sistema de restricciones para llevar a cabo el cálculo de temperatura descrito anteriormente, primero creamos dos conectores, `C` y `F`, llamando al constructor `make/connector`, y vinculamos `C` y `F` en una red apropiada:

(define C (make-connector)) (define F (make-connector)) (celsius-fahrenheit-converter C F) *ok*

El procedimiento que crea la red se define de la siguiente manera:

(define (celsius-fahrenheit-converter c f) (let ((u (make-connector)) (v (make-connector)) (w (make-connector)) (x (make-connector)) (y (make-connector))) (multiplier c w u) (multiplier v x u) (adder v y f) (constant 9 w) (constant 5 x) (constant 32 y) 'ok))

Este procedimiento crea los conectores internos `u`, `v`, `w`, `x`, y `y`, y los enlaza como se muestra en la Figura 3.28 usando los constructores de restricciones primitivas `adder`, `multiplier`, y `constant`. Al igual que con el simulador de circuitos digitales de Section 3.3.4, expresar estas combinaciones de elementos primitivos en términos de procedimientos proporciona automáticamente a nuestro lenguaje un medio de abstracción para objetos compuestos.

Para ver la red en acción, podemos colocar sondas en los conectores `C` y `F`, usando un procedimiento `probe` similar al que usamos para monitorear los cables en Section 3.3.4. Colocar una sonda en un conector hará que se imprima un mensaje cada vez que se le dé un valor al conector:

(probe "Temperatura en Celsius" C) (probe "Temperatura en Fahrenheit" F)

A continuación, establecemos el valor de `C` en 25. (El tercer argumento de `set/value!` le dice a `C` que esta directiva proviene del `usuario`.)

(set-value! C 25 'user) *Sonda: Temperatura Celsius = 25 Sonda: Temperatura*

Fahrenheit = 77 hecho

La sonda en `C` se despierta e informa el valor. `C` también propaga su valor a través de la red como se describió anteriormente. Esto establece `F` en 77, lo cual es informado por la sonda en `F`.

Ahora podemos intentar establecer `F` en un nuevo valor, digamos 212:

(set-value! F 212 'user) *¡Error! Contradicción (77 212)*

El conector se queja de que ha detectado una contradicción: su valor es 77, y alguien está intentando establecerlo en 212. Si realmente queremos reutilizar la red con nuevos valores, podemos decirle a `C` que olvide su antiguo valor:

(forget-value! C 'user) *Sonda: Temperatura Celsius = ? Sonda: Temperatura Fahrenheit = ? hecho*

`C` descubre que el `usuario`, que estableció su valor originalmente, ahora está retractando ese valor, por lo que `C` acepta perder su valor, como muestra la sonda, e informa al resto de la red de este hecho. Esta información eventualmente se propaga a `F`, que ahora descubre que no tiene ninguna razón para seguir creyendo que su propio valor es 77. Por lo tanto, `F` también renuncia a su valor, como muestra la sonda.

Ahora que `F` no tiene valor, somos libres de establecerlo en 212:

(set-value! F 212 'user) *Sonda: Temperatura Fahrenheit = 212 Sonda: Temperatura Celsius = 100 hecho*

Este nuevo valor, al propagarse por la red, obliga a `C` a tener un valor de 100, y esto es registrado por la sonda en `C`. Nótese que se está utilizando la misma red para calcular `C` dado `F` y para calcular `F` dado `C`. Esta no direccionalidad del cálculo es la característica distintiva de los sistemas basados en restricciones.

Implementando el sistema de restricciones

El sistema de restricciones se implementa mediante objetos procedimentales con estado local, de una manera muy similar al simulador de circuitos digitales de Section 3.3.4. Aunque los objetos primitivos del sistema de restricciones son algo más complejos, el sistema general es más simple, ya que no hay preocupación por agendas y retardos lógicos.

Las operaciones básicas sobre los conectores son las siguientes:

- `(has/value? ⟨conector⟩)` indica si el conector tiene un valor.
- `(get/value ⟨conector⟩)` devuelve el valor actual del conector.
- `(set/value! ⟨conector⟩ ⟨nuevo/valor⟩ ⟨informante⟩)` indica que el informante está solicitando al conector que establezca su valor en el nuevo valor.
- `(forget/value! ⟨conector⟩ ⟨retractor⟩)` le dice al conector que el retractor le está solicitando que olvide su valor.
- `(connect ⟨conector⟩ ⟨nueva/restricción⟩)` le dice al conector que participe en la nueva restricción.

Los conectores se comunican con las restricciones por medio de los procedimientos `inform/about/value`, que le dice a la restricción dada que el conector tiene un valor, y `inform/about/no/value`, que le dice a la restricción que el conector ha perdido su valor.

`adder` construye una restricción de sumador entre los conectores sumandos `a1` y `a2` y un conector `sum`. Un sumador se implementa como un procedimiento con estado local (el procedimiento `me` a continuación):

(define (adder a1 a2 sum) (define (process-new-value) (cond ((and (has-value? a1) (has-value? a2)) (set-value! sum (+ (get-value a1) (get-value a2)) me)) ((and (has-value? a1) (has-value? sum)) (set-value! a2 (- (get-value sum) (get-value a1)) me)) ((and (has-value? a2) (has-value? sum)) (set-value! a1 (- (get-value sum) (get-value a2)) me)))) (define (process-forget-value) (forget-value! sum me) (forget-value! a1 me) (forget-value! a2 me) (process-new-value)) (define (me request) (cond ((eq? request 'I-have-a-value) (process-new-value)) ((eq? request 'I-lost-my-value) (process-forget-value)) (else (error "Unknown request: ADDER" request)))) (connect a1 me) (connect a2 me) (connect sum me) me)

`adder` conecta el nuevo sumador a los conectores designados y lo devuelve como su valor. El procedimiento `me`, que representa el sumador, actúa como un despacho a los procedimientos locales. Las siguientes "interfaces de sintaxis" (ver Nota 27 en Section 3.3.4) se utilizan junto con el despacho:

(define (inform-about-value constraint) (constraint 'I-have-a-value)) (define (inform-about-no-value constraint) (constraint 'I-lost-my-value))

El procedimiento local del sumador `process/new/value` se llama cuando se informa al sumador que uno de sus conectores tiene un valor. El sumador primero comprueba si tanto `a1` como `a2` tienen valores. Si es así, le dice a

`sum` que establezca su valor en la suma de los dos sumandos. El argumento `informant` de `set/value!` es `me`, que es el propio objeto sumador. Si `a1` y `a2` no tienen ambos valores, entonces el sumador comprueba si quizás `a1` y `sum` tienen valores. Si es así, establece `a2` en la diferencia de estos dos. Finalmente, si `a2` y `sum` tienen valores, esto le da al sumador suficiente información para establecer `a1`. Si se le dice al sumador que uno de sus conectores ha perdido un valor, solicita que todos sus conectores pierdan ahora sus valores. (Solo se pierden los valores que fueron establecidos por este sumador.) Luego ejecuta `process/new/value`. La razón de este último paso es que uno o más conectores todavía pueden tener un valor (es decir, un conector puede haber tenido un valor que no fue establecido originalmente por el sumador), y estos valores pueden necesitar ser propagados de nuevo a través del sumador.

Un multiplicador es muy similar a un sumador. Establecerá su `product` en 0 si cualquiera de los factores es 0, incluso si el otro factor no se conoce.

(define (multiplier m1 m2 product) (define (process-new-value) (cond ((or (and (has-value? m1) (= (get-value m1) 0)) (and (has-value? m2) (= (get-value m2) 0))) (set-value! product 0 me)) ((and (has-value? m1) (has-value? m2)) (set-value! product (* (get-value m1) (get-value m2)) me)) ((and (has-value? product) (has-value? m1)) (set-value! m2 (/ (get-value product) (get-value m1)) me)) ((and (has-value? product) (has-value? m2)) (set-value! m1 (/ (get-value product) (get-value m2)) me)))) (define (process-forget-value) (forget-value! product me) (forget-value! m1 me) (forget-value! m2 me) (process-new-value)) (define (me request) (cond ((eq? request 'I-have-a-value) (process-new-value)) ((eq? request 'I-lost-my-value) (process-forget-value)) (else (error "Unknown request: MULTIPLIER" request)))) (connect m1 me) (connect m2 me) (connect product me) me)

Un constructor `constante` simplemente establece el valor del conector designado. Cualquier mensaje `I/have/a/value` o `I/lost/my/value` enviado a la caja constante producirá un error.

(define (constant value connector) (define (me request) (error "Unknown request: CONSTANT" request)) (connect connector me) (set-value! connector value me) me)

Finalmente, una sonda imprime un mensaje sobre el establecimiento o desestablecimiento del conector designado:

(define (probe name connector) (define (print-probe value) (newline) (display "Probe: ") (display name) (display " = ") (display value)) (define (process-new-value) (print-probe (get-value connector))) (define (process-

forget-value) (print-probe "?")) (define (me request) (cond ((eq? request 'I-have-a-value) (process-new-value)) ((eq? request 'I-lost-my-value) (process-forget-value)) (else (error "Unknown request: PROBE" request)))) (connect connector me) me)

Representando conectores

Un conector se representa como un objeto procedimental con variables de estado locales `value`, el valor actual del conector; `informant`, el objeto que estableció el valor del conector; y `constraints`, una lista de las restricciones en las que participa el conector.

(define (make-connector) (let ((value false) (informant false) (constraints '())) (define (set-my-value newval setter) (cond ((not (has-value? me)) (set! value newval) (set! informant setter) (for-each-except setter inform-about-value constraints)) ((not (= value newval)) (error "Contradiction" (list value newval))) (else 'ignored))) (define (forget-my-value retractor) (if (eq? retractor informant) (begin (set! informant false) (for-each-except retractor inform-about-no-value constraints)) 'ignored)) (define (connect new-constraint) (if (not (memq new-constraint constraints)) (set! constraints (cons new-constraint constraints))) (if (has-value? me) (inform-about-value new-constraint)) 'done) (define (me request) (cond ((eq? request 'has-value?) (if informant true false)) ((eq? request 'value) value) ((eq? request 'set-value!) set-my-value) ((eq? request 'forget) forget-my-value) ((eq? request 'connect) connect) (else (error "Unknown operation: CONNECTOR" request)))) me))

El procedimiento local del conector `set/my/value` se llama cuando hay una solicitud para establecer el valor del conector. Si el conector no tiene actualmente un valor, establecerá su valor y recordará como `informant` la restricción que solicitó que se estableciera el valor.[32] Luego, el conector notificará a todas sus restricciones participantes excepto a la restricción que solicitó que se estableciera el valor. Esto se logra usando el siguiente iterador, que aplica un procedimiento designado a todos los elementos de una lista excepto a uno dado:

(define (for-each-except exception procedure list) (define (loop items) (cond ((null? items) 'done) ((eq? (car items) exception) (loop (cdr items))) (else (procedure (car items)) (loop (cdr items))))) (loop list))

Si se le pide a un conector que olvide su valor, ejecuta el procedimiento local `forget/my/value`, que primero comprueba para asegurarse de que la solicitud proviene del mismo objeto que estableció el valor originalmente. Si es así, el conector informa a sus restricciones asociadas sobre la pérdida del valor.

El procedimiento local `connect` agrega la nueva restricción designada a la lista de restricciones si aún no está en esa lista. Luego, si el conector tiene un valor, informa a la nueva restricción de este hecho.

El procedimiento `me` del conector sirve como un despachador para los otros procedimientos internos y también representa el conector como un objeto. Los siguientes procedimientos proporcionan una interfaz de sintaxis para el despachador:

(define (has-value? connector) (connector 'has-value?)) (define (get-value connector) (connector 'value)) (define (set-value! connector new-value informant) ((connector 'set-value!) new-value informant)) (define (forget-value! connector retractor) ((connector 'forget) retractor)) (define (connect connector new-constraint) ((connector 'connect) new-constraint))

Ejercicio 3.33: Usando restricciones primitivas de multiplicador, sumador y constante, defina un procedimiento `averager` que tome tres conectores `a`, `b` y `c` como entradas y establezca la restricción de que el valor de `c` es el promedio de los valores de `a` y `b`.

Ejercicio 3.34: Louis Reasoner quiere construir un cuadrador, un dispositivo de restricción con dos terminales tal que el valor del conector `b` en el segundo terminal siempre será el cuadrado del valor `a` en el primer terminal. Él propone el siguiente dispositivo simple hecho de un multiplicador:

(define (squarer a b) (multiplier a a b))

Hay un grave defecto en esta idea. Explique.

Ejercicio 3.35: Ben Bitdiddle le dice a Louis que una forma de evitar el problema del Ejercicio 3.34 es definir un cuadrador como una nueva restricción primitiva. Complete las porciones faltantes en el esquema de Ben para un procedimiento que implemente tal restricción:

(define (squarer a b) (define (process-new-value) (if (has-value? b) (if (< (get-value b) 0) (error "square less than 0: SQUARER" (get-value b)) < *alternativa1* >) < *alternativa2* >)) (define (process-forget-value) < *cuerpo1* >) (define (me request) < *cuerpo2* >) < *resto de la definición* > me)

Ejercicio 3.36: Supongamos que evaluamos la siguiente secuencia de expresiones en el entorno global:

(define a (make-connector)) (define b (make-connector)) (set-value! a 10

'user)

En algún momento durante la evaluación de `set/value!`, la siguiente expresión del procedimiento local del conector se evalúa:

(for-each-except setter inform-about-value constraints)

Dibuja un diagrama de entorno que muestre el entorno en el que se evalúa la expresión anterior.

Ejercicio 3.37: El procedimiento `celsius/fahrenheit/converter` es engorroso en comparación con un estilo de definición más orientado a expresiones, como

(define (celsius-fahrenheit-converter x) (c+ (c* (c/ (cv 9) (cv 5)) x) (cv 32))) (define C (make-connector)) (define F (celsius-fahrenheit-converter C))

Aquí `c+`, `c*`, etc. son las versiones de "restricción" de las operaciones aritméticas. Por ejemplo, `c+` toma dos conectores como argumentos y devuelve un conector que está relacionado con estos por una restricción de sumador:

(define (c+ x y) (let ((z (make-connector))) (adder x y z) z))

Defina procedimientos análogos `c-`, `c*`, `c/`, y `cv` (valor constante) que nos permitan definir restricciones compuestas como en el ejemplo del convertidor anterior.[33]

Concurrencia: El Tiempo es Esencial

Hemos visto el poder de los objetos computacionales con estado local como herramientas para modelar. Sin embargo, como advirtió Section 3.1.3, este poder tiene un precio: la pérdida de la transparencia referencial, que da lugar a un laberinto de preguntas sobre la identidad y el cambio, y la necesidad de abandonar el modelo de evaluación por sustitución en favor del modelo de entorno, más intrincado.

El problema central que subyace a la complejidad del estado, la identidad y el cambio es que al introducir la asignación nos vemos obligados a admitir el *tiempo* en nuestros modelos computacionales. Antes de introducir la asignación, todos nuestros programas eran atemporales, en el sentido de que cualquier expresión que tiene un valor siempre tiene el mismo valor. En contraste, recordemos el ejemplo de modelar retiros de una cuenta bancaria y

devolver el saldo resultante, introducido al principio de Section 3.1.1:

(withdraw 25) *75* (withdraw 25) *50*

Aquí, las evaluaciones sucesivas de la misma expresión producen valores diferentes. Este comportamiento surge del hecho de que la ejecución de sentencias de asignación (en este caso, asignaciones a la variable `balance`) delinea *momentos en el tiempo* en los que los valores cambian. El resultado de evaluar una expresión depende no solo de la expresión en sí, sino también de si la evaluación ocurre antes o después de estos momentos. Construir modelos en términos de objetos computacionales con estado local nos obliga a confrontar el tiempo como un concepto esencial en la programación.

Podemos ir más allá en la estructuración de modelos computacionales para que coincidan con nuestra percepción del mundo físico. Los objetos en el mundo no cambian uno a la vez en secuencia. Más bien los percibimos como actuando *concurrentemente*—todos a la vez. Por lo tanto, a menudo es natural modelar sistemas como colecciones de procesos computacionales que se ejecutan concurrentemente. Así como podemos hacer que nuestros programas sean modulares organizando modelos en términos de objetos con estado local separado, a menudo es apropiado dividir los modelos computacionales en partes que evolucionan por separado y concurrentemente. Incluso si los programas se van a ejecutar en una computadora secuencial, la práctica de escribir programas como si fueran a ejecutarse concurrentemente obliga al programador a evitar restricciones de tiempo no esenciales y, por lo tanto, hace que los programas sean más modulares.

Además de hacer los programas más modulares, la computación concurrente puede proporcionar una ventaja de velocidad sobre la computación secuencial. Las computadoras secuenciales ejecutan solo una operación a la vez, por lo que la cantidad de tiempo que se tarda en realizar una tarea es proporcional al número total de operaciones realizadas.[34] Sin embargo, si es posible descomponer un problema en partes que son relativamente independientes y necesitan comunicarse solo raramente, puede ser posible asignar las partes a procesadores de cómputo separados, produciendo una ventaja de velocidad proporcional al número de procesadores disponibles.

Desafortunadamente, las complejidades introducidas por la asignación se vuelven aún más problemáticas en presencia de concurrencia. El hecho de la ejecución concurrente, ya sea porque el mundo opera en paralelo o porque nuestras computadoras lo hacen, implica una complejidad adicional en nuestra comprensión del tiempo.

La Naturaleza del Tiempo en Sistemas Concurrentes

A primera vista, el tiempo parece sencillo. Es un orden impuesto a los eventos.[35] Para cualquier evento *A* y *B*, o *A* ocurre antes que *B*, *A* y *B* son simultáneos, o *A* ocurre después que *B*. Por ejemplo, volviendo al ejemplo de la cuenta bancaria, supongamos que Peter retira $10 y Paul retira $25 de una cuenta conjunta que inicialmente contiene $100, dejando $65 en la cuenta. Dependiendo del orden de los dos retiros, la secuencia de saldos en la cuenta es $100 → $90 → $65 o $100 → $75 → $65. En una implementación informática del sistema bancario, esta secuencia cambiante de saldos podría modelarse mediante asignaciones sucesivas a una variable `balance`.

En situaciones complejas, sin embargo, tal visión puede ser problemática. Supongamos que Peter y Paul, y otras personas además, están accediendo a la misma cuenta bancaria a través de una red de cajeros automáticos distribuidos por todo el mundo. La secuencia real de saldos en la cuenta dependerá críticamente de la temporización detallada de los accesos y de los detalles de la comunicación entre las máquinas.

Esta indeterminación en el orden de los eventos puede plantear serios problemas en el diseño de sistemas concurrentes. Por ejemplo, supongamos que los retiros hechos por Peter y Paul se implementan como dos procesos separados que comparten una variable común `balance`, cada proceso especificado por el procedimiento dado en Section 3.1.1:

(define (withdraw amount) (if (>= balance amount) (begin (set! balance (- balance amount)) balance) "Fondos insuficientes"))

Si los dos procesos operan de forma independiente, entonces Peter podría probar el saldo e intentar retirar una cantidad legítima. Sin embargo, Paul podría retirar algunos fondos entre el momento en que Peter comprueba el saldo y el momento en que Peter completa el retiro, invalidando así la prueba de Peter.

Las cosas pueden ser aún peores. Considere la expresión

(set! balance (- balance amount))

ejecutado como parte de cada proceso de retiro. Este consta de tres pasos: (1) acceder al valor de la variable `balance`; (2) calcular el nuevo saldo; (3) establecer `balance` a este nuevo valor. Si los retiros de Peter y Paul ejecutan esta declaración concurrentemente, entonces los dos retiros podrían entrelazar el orden en que acceden a `balance` y lo establecen al nuevo valor.

Figura 3.29: Diagrama de tiempo que muestra cómo el entrelazamiento

del orden de los eventos en dos retiros bancarios puede llevar a un saldo final incorrecto.

El diagrama de tiempo en la Figura 3.29 representa un orden de eventos donde `balance` comienza en 100, Pedro retira 10, Pablo retira 25, y sin embargo el valor final de `balance` es 75. Como se muestra en el diagrama, la razón de esta anomalía es que la asignación de 75 de Pablo a `balance` se hace bajo la suposición de que el valor de `balance` a decrementar es 100. Esa suposición, sin embargo, se volvió inválida cuando Pedro cambió `balance` a 90. Esto es un fallo catastrófico para el sistema bancario, porque la cantidad total de dinero en el sistema no se conserva. Antes de las transacciones, la cantidad total de dinero era de $100. Después, Pedro tiene $10, Pablo tiene $25, y el banco tiene $75.[36]

El fenómeno general ilustrado aquí es que varios procesos pueden compartir una variable de estado común. Lo que lo complica es que más de un proceso puede estar intentando manipular el estado compartido al mismo tiempo. Para el ejemplo de la cuenta bancaria, durante cada transacción, cada cliente debería poder actuar como si los demás clientes no existieran. Cuando un cliente cambia el saldo de una manera que depende del saldo, debe poder asumir que, justo antes del momento del cambio, el saldo sigue siendo el que pensaba que era.

Comportamiento correcto de programas concurrentes

El ejemplo anterior tipifica los errores sutiles que pueden colarse en los programas concurrentes. La raíz de esta complejidad reside en las asignaciones a variables que son compartidas entre los diferentes procesos. Ya sabemos que debemos tener cuidado al escribir programas que usan `set!`, porque los resultados de un cálculo dependen del orden en que ocurren las asignaciones.[37] Con procesos concurrentes debemos tener especial cuidado con las asignaciones, porque es posible que no podamos controlar el orden de las asignaciones realizadas por los diferentes procesos. Si varios de estos cambios pudieran hacerse concurrentemente (como con dos depositantes accediendo a una cuenta conjunta) necesitamos alguna forma de asegurar que nuestro sistema se comporte correctamente. Por ejemplo, en el caso de retiros de una cuenta bancaria conjunta, debemos asegurar que el dinero se conserve. Para que los programas concurrentes se comporten correctamente, es posible que tengamos que poner algunas restricciones a la ejecución concurrente.

Una posible restricción sobre la concurrencia estipularía que no pueden ocurrir dos operaciones que cambien ninguna variable de estado compartida al mismo tiempo. Este es un requisito extremadamente estricto. Para la banca

distribuida, requeriría que el diseñador del sistema asegurara que solo una transacción pudiera proceder a la vez. Esto sería tanto ineficiente como demasiado conservador. La Figura 3.30 muestra a Peter y Paul compartiendo una cuenta bancaria, donde Paul tiene también una cuenta privada. El diagrama ilustra dos retiros de la cuenta compartida (uno por Peter y otro por Paul) y un depósito en la cuenta privada de Paul.[38] Los dos retiros de la cuenta compartida no deben ser concurrentes (ya que ambos acceden y actualizan la misma cuenta), y el depósito y el retiro de Paul no deben ser concurrentes (ya que ambos acceden y actualizan la cantidad en la billetera de Paul). Pero no debería haber ningún problema en permitir que el depósito de Paul en su cuenta privada proceda concurrentemente con el retiro de Peter de la cuenta compartida.

Figura 3.30: Depósitos y retiros concurrentes de una cuenta conjunta account en el Banco1 y una cuenta privada en el Banco2.

Una restricción menos estricta sobre la concurrencia aseguraría que un sistema concurrente produzca el mismo resultado que si los procesos se hubieran ejecutado secuencialmente en algún orden. Hay dos aspectos importantes en este requisito. Primero, no requiere que los procesos se ejecuten realmente secuencialmente, sino solo que produzcan resultados que sean los mismos *como si* se hubieran ejecutado secuencialmente. Para el ejemplo de la Figura 3.30, el diseñador del sistema de cuentas bancarias puede permitir con seguridad que el depósito de Paul y el retiro de Peter ocurran concurrentemente, porque el resultado neto será el mismo que si las dos operaciones hubieran ocurrido secuencialmente. Segundo, puede haber más de un resultado "correcto" posible producido por un programa concurrente, porque solo requerimos que el resultado sea el mismo que para *algún* orden secuencial. Por ejemplo, supongamos que la cuenta conjunta de Peter y Paul comienza con $100, y Peter deposita $40 mientras Paul retira concurrentemente la mitad del dinero de la cuenta. Entonces la ejecución secuencial podría resultar en que el saldo de la cuenta sea de $70 o $90 (ver Ejercicio 3.38).[39]

Existen requisitos aún más débiles para la ejecución correcta de programas concurrentes. Un programa para simular la difusión (digamos, el flujo de calor en un objeto) podría consistir en un gran número de procesos, cada uno representando un pequeño volumen de espacio, que actualizan sus valores concurrentemente. Cada proceso cambia repetidamente su valor al promedio de su propio valor y los valores de sus vecinos. Este algoritmo converge a la respuesta correcta independientemente del orden en que se realicen las operaciones; no hay necesidad de ninguna restricción sobre el uso

concurrente de los valores compartidos.

Ejercicio 3.38: Suponga que Peter, Paul y Mary comparten una cuenta bancaria conjunta que inicialmente contiene $100. Concurrentemente, Peter deposita $10, Paul retira $20 y Mary retira la mitad del dinero de la cuenta, ejecutando los siguientes comandos:

Pedro: (set! balance (+ balance 10)) Pablo: (set! balance (- balance 20)) María: (set! balance (- balance (/ balance 2)))

1. Enumere todos los diferentes valores posibles para `balance` después de que estas tres transacciones se hayan completado, asumiendo que el sistema bancario obliga a los tres procesos a ejecutarse secuencialmente en algún orden.

2. ¿Cuáles son algunos otros valores que podrían producirse si el sistema permite que los procesos se intercalen? Dibuja diagramas de tiempo como el de la Figura 3.29 para explicar cómo pueden ocurrir estos valores.

Mecanismos para Controlar la Concurrencia

Hemos visto que la dificultad de tratar con procesos concurrentes radica en la necesidad de considerar el entrelazamiento del orden de los eventos en los diferentes procesos. Por ejemplo, supongamos que tenemos dos procesos, uno con tres eventos ordenados (a, b, c) y otro con tres eventos ordenados (x, y, z). Si los dos procesos se ejecutan concurrentemente, sin restricciones sobre cómo se entrelaza su ejecución, entonces hay 20 ordenamientos diferentes posibles para los eventos que son consistentes con los ordenamientos individuales para los dos procesos:

(a,b,c,x,y,z) (a,x,b,y,c,z) (x,a,b,c,y,z) (x,a,y,z,b,c) (a,b,x,c,y,z) (a,x,b,y,z,c) (x,a,b,y,c,z) (x,y,a,b,c,z) (a,b,x,y,c,z) (a,x,y,b,c,z) (x,a,b,y,z,c) (x,y,a,b,z,c) (a,b,x,y,z,c) (a,x,y,b,z,c) (x,a,y,b,c,z) (x,y,a,z,b,c) (a,x,b,c,y,z) (a,x,y,z,b,c) (x,a,y,b,z,c) (x,y,z,a,b,c)

Como programadores diseñando este sistema, tendríamos que considerar los efectos de cada uno de estos 20 ordenamientos y verificar que cada comportamiento es aceptable. Tal enfoque se vuelve rápidamente inmanejable a medida que aumentan los números de procesos y eventos.

Un enfoque más práctico para el diseño de sistemas concurrentes es idear mecanismos generales que nos permitan restringir el entrelazamiento de procesos concurrentes para que podamos estar seguros de que el

comportamiento del programa es correcto. Se han desarrollado muchos mecanismos para este propósito. En esta sección, describimos uno de ellos, el *serializador*.

Serialización del acceso al estado compartido

La serialización implementa la siguiente idea: los procesos se ejecutarán concurrentemente, pero habrá ciertas colecciones de procedimientos que no podrán ejecutarse concurrentemente. Más precisamente, la serialización crea conjuntos distinguidos de procedimientos de tal manera que solo se permite que ocurra una ejecución de un procedimiento en cada conjunto serializado a la vez. Si algún procedimiento en el conjunto está siendo ejecutado, entonces un proceso que intente ejecutar cualquier procedimiento en el conjunto se verá obligado a esperar hasta que la primera ejecución haya terminado.

Podemos usar la serialización para controlar el acceso a variables compartidas. Por ejemplo, si queremos actualizar una variable compartida basándonos en el valor anterior de esa variable, ponemos el acceso al valor anterior de la variable y la asignación del nuevo valor a la variable en el mismo procedimiento. Luego nos aseguramos de que ningún otro procedimiento que asigne a la variable pueda ejecutarse concurrentemente con este procedimiento serializando todos estos procedimientos con el mismo serializador. Esto garantiza que el valor de la variable no pueda ser cambiado entre un acceso y la asignación correspondiente.

Serializadores en Scheme

Para hacer más concreto el mecanismo anterior, supongamos que hemos extendido Scheme para incluir un procedimiento llamado `parallel/execute`:

(parallel-execute $\langle$ *p* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$ $\langle$ *p* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$ … $\langle$ *p* $\color{SchemeDark}_{\hbox{\ttfamily\itshape \scriptsize k}}\rangle$)

Cada ⟨*p*⟩ debe ser un procedimiento sin argumentos. `parallel/execute` crea un proceso separado para cada ⟨*p*⟩, que aplica ⟨*p*⟩ (sin argumentos). Todos estos procesos se ejecutan concurrentemente.[40]

Como ejemplo de cómo se usa esto, considere

(define x 10) (parallel-execute (lambda () (set! x (* x x))) (lambda () (set! x (+ x 1))))

Esto crea dos procesos concurrentes—P_1, que establece `x` en `x` por `x`, y P_2, que incrementa `x`. Una vez completada la ejecución, `x` quedará con uno de cinco valores posibles, dependiendo del entrelazamiento de los eventos de P_1 y P_2:

101: P_1 establece `x` en 100 y luego P_2 incrementa `x` a 101. 121: P_2 incrementa `x` a 11 y luego P_1 establece `x` en `x * x`. 110: P_2 cambia `x` de 10 a 11 entre las dos veces que P_1 accede al valor de `x` durante la evaluación de `(* x x)`. 11: P_2 accede a `x`, luego P_1 establece `x` en 100, luego P_2 establece `x`. 100: P_1 accede a `x` (dos veces), luego P_2 establece `x` en 11, luego P_1 establece `x`.

Podemos restringir la concurrencia usando procedimientos serializados, que son creados por *serializadores*. Los serializadores se construyen con `make/serializer`, cuya implementación se da a continuación. Un serializador toma un procedimiento como argumento y devuelve un procedimiento serializado que se comporta como el procedimiento original. Todas las llamadas a un serializador dado devuelven procedimientos serializados en el mismo conjunto.

Así, en contraste con el ejemplo anterior, ejecutar

(define x 10) (define s (make-serializer)) (parallel-execute (s (lambda () (set! x (* x x)))) (s (lambda () (set! x (+ x 1)))))

solo puede producir dos valores posibles para `x`, 101 o 121. Las otras posibilidades se eliminan, porque la ejecución de P_1 y P_2 no puede entrelazarse.

Aquí hay una versión del procedimiento `make/account` de Section 3.1.1, donde los depósitos y retiros han sido serializados:

(define (make-account balance) (define (withdraw amount) (if (>= balance amount) (begin (set! balance (- balance amount)) balance) "Insufficient funds")) (define (deposit amount) (set! balance (+ balance amount)) balance) (let ((protected (make-serializer))) (define (dispatch m) (cond ((eq? m 'withdraw) (protected withdraw)) ((eq? m 'deposit) (protected deposit)) ((eq? m 'balance) balance) (else (error "Unknown request: MAKE-ACCOUNT" m)))) dispatch))

Con esta implementación, dos procesos no pueden retirar o depositar en una sola cuenta concurrentemente. Esto elimina la fuente del error ilustrado en la Figura 3.29, donde Pedro cambia el saldo de la cuenta entre los momentos en que Pablo accede al saldo para calcular el nuevo valor y cuando Pablo realmente realiza la asignación. Por otro lado, cada cuenta tiene su propio

serializador, de modo que los depósitos y retiros para diferentes cuentas pueden proceder concurrentemente.

Ejercicio 3.39: ¿Cuáles de las cinco posibilidades en la ejecución paralela mostrada anteriormente permanecen si en cambio serializamos la ejecución de la siguiente manera?

(define x 10) (define s (make-serializer)) (parallel-execute (lambda () (set! x ((s (lambda () (* x x)))))) (s (lambda () (set! x (+ x 1))))))

Ejercicio 3.40: Dé todos los valores posibles de `x` que pueden resultar de la ejecución

(define x 10) (parallel-execute (lambda () (set! x (* x x))) (lambda () (set! x (* x x x))))

¿Cuáles de estas posibilidades permanecen si en su lugar usamos procedimientos serializados?

(define x 10) (define s (make-serializer)) (parallel-execute (s (lambda () (set! x (* x x)))) (s (lambda () (set! x (* x x x)))))

Ejercicio 3.41: A Ben Bitdiddle le preocupa que sería mejor implementar la cuenta bancaria de la siguiente manera (donde la línea comentada ha sido cambiada):

(define (make-account balance) (define (withdraw amount) (if (>= balance amount) (begin (set! balance (- balance amount)) balance) "Fondos insuficientes")) (define (deposit amount) (set! balance (+ balance amount)) balance) (let ((protected (make-serializer))) (define (dispatch m) (cond ((eq? m 'withdraw) (protected withdraw)) ((eq? m 'deposit) (protected deposit)) ((eq? m 'balance) ((protected (lambda () balance)))) ; serializado (else (error "Solicitud desconocida: MAKE-ACCOUNT" m)))) dispatch))

porque permitir el acceso no serializado al saldo bancario puede resultar en un comportamiento anómalo. ¿Estás de acuerdo? ¿Hay algún escenario que demuestre la preocupación de Ben?

Ejercicio 3.42: Ben Bitdiddle sugiere que es una pérdida de tiempo crear un nuevo procedimiento serializado en respuesta a cada mensaje de `withdraw` y `deposit`. Dice que `make/account` podría cambiarse para que las llamadas a `protected` se hagan fuera del procedimiento `dispatch`. Es decir, una cuenta devolvería el mismo procedimiento serializado (que se creó al mismo tiempo que la cuenta) cada vez que se le solicite un

procedimiento de retiro.

(define (make-account balance) (define (withdraw amount) (if (>= balance amount) (begin (set! balance (- balance amount)) balance) "Fondos insuficientes")) (define (deposit amount) (set! balance (+ balance amount)) balance) (let ((protected (make-serializer))) (let ((protected-withdraw (protected withdraw)) (protected-deposit (protected deposit))) (define (dispatch m) (cond ((eq? m 'withdraw) protected-withdraw) ((eq? m 'deposit) protected-deposit) ((eq? m 'balance) balance) (else (error "Solicitud desconocida: MAKE-ACCOUNT" m)))) dispatch)))

¿Es este un cambio seguro de hacer? En particular, ¿hay alguna diferencia en la concurrencia permitida por estas dos versiones de `make/account`?

Complejidad del uso de múltiples recursos compartidos

Los serializadores proporcionan una poderosa abstracción que ayuda a aislar las complejidades de los programas concurrentes para que puedan ser tratados con cuidado y (con suerte) correctamente. Sin embargo, aunque el uso de serializadores es relativamente sencillo cuando hay un solo recurso compartido (como una sola cuenta bancaria), la programación concurrente puede ser traicioneramente difícil cuando hay múltiples recursos compartidos.

Para ilustrar una de las dificultades que pueden surgir, supongamos que deseamos intercambiar los saldos de dos cuentas bancarias. Accedemos a cada cuenta para encontrar el saldo, calculamos la diferencia entre los saldos, retiramos esta diferencia de una cuenta y la depositamos en la otra. Podríamos implementar esto de la siguiente manera:[41]

(define (exchange account1 account2) (let ((difference (- (account1 'balance) (account2 'balance)))) ((account1 'withdraw) difference) ((account2 'deposit) difference)))

Este procedimiento funciona bien cuando solo un proceso está tratando de hacer el intercambio. Supongamos, sin embargo, que Pedro y Pablo tienen acceso a las cuentas $a1$, $a2$ y $a3$, y que Pedro intercambia $a1$ y $a2$ mientras Pablo intercambia concurrentemente $a1$ y $a3$. Incluso con los depósitos y retiros de las cuentas serializados para cuentas individuales (como en el procedimiento `make/account` mostrado anteriormente en esta sección), `exchange` todavía puede producir resultados incorrectos. Por ejemplo, Pedro podría calcular la diferencia en los saldos de $a1$ y $a2$, pero luego Pablo podría

cambiar el saldo en *a*1 antes de que Pedro pueda completar el intercambio.[42] Para un comportamiento correcto, debemos hacer arreglos para que el procedimiento `exchange` bloquee cualquier otro acceso concurrente a las cuentas durante todo el tiempo del intercambio.

Una forma de lograr esto es usando los serializadores de ambas cuentas para serializar todo el procedimiento `exchange`. Para hacer esto, haremos arreglos para acceder al serializador de una cuenta. Nótese que estamos rompiendo deliberadamente la modularidad del objeto de cuenta bancaria al exponer el serializador. La siguiente versión de `make/account` es idéntica a la versión original dada en Section 3.1.1, excepto que se proporciona un serializador para proteger la variable de saldo, y el serializador se exporta a través del paso de mensajes:

(define (make-account-and-serializer balance) (define (withdraw amount) (if (> = balance amount) (begin (set! balance (- balance amount)) balance) "Fondos insuficientes")) (define (deposit amount) (set! balance (+ balance amount)) balance) (let ((balance-serializer (make-serializer))) (define (dispatch m) (cond ((eq? m 'withdraw) withdraw) ((eq? m 'deposit) deposit) ((eq? m 'balance) balance) ((eq? m 'serializer) balance-serializer) (else (error "Unknown request: MAKE-ACCOUNT" m)))) dispatch))

Podemos usar esto para hacer depósitos y retiros serializados. Sin embargo, a diferencia de nuestra cuenta serializada anterior, ahora es responsabilidad de cada usuario de objetos de cuenta bancaria gestionar explícitamente la serialización, por ejemplo, de la siguiente manera:[43]

(define (deposit account amount) (let ((s (account 'serializer)) (d (account 'deposit))) ((s d) amount)))

Exportar el serializador de esta manera nos da suficiente flexibilidad para implementar un programa de intercambio serializado. Simplemente serializamos el procedimiento `exchange` original con los serializadores de ambas cuentas:

(define (serialized-exchange account1 account2) (let ((serializer1 (account1 'serializer)) (serializer2 (account2 'serializer))) ((serializer1 (serializer2 exchange)) account1 account2)))

> **Ejercicio 3.43:** Supongamos que los saldos de tres cuentas comienzan en $10, $20 y $30, y que se ejecutan múltiples procesos, intercambiando los saldos de las cuentas. Argumente que si los procesos se ejecutan secuencialmente, después de cualquier número de intercambios concurrentes, los saldos de las cuentas deberían ser $10, $20 y $30 en algún orden. Dibuje un diagrama de tiempo como el de la Figura 3.29

para mostrar cómo esta condición puede ser violada si los intercambios se implementan usando la primera versión del programa de intercambio de cuentas en esta sección. Por otro lado, argumente que incluso con este programa `exchange`, la suma de los saldos de las cuentas se preservará. Dibuje un diagrama de tiempo para mostrar cómo incluso esta condición sería violada si no serializáramos las transacciones en cuentas individuales.

Ejercicio 3.44: Considere el problema de transferir una cantidad de una cuenta a otra. Ben Bitdiddle afirma que esto se puede lograr con el siguiente procedimiento, incluso si hay varias personas transfiriendo dinero concurrentemente entre varias cuentas, usando cualquier mecanismo de cuenta que serialice las transacciones de depósito y retiro, por ejemplo, la versión de `make/account` en el texto anterior.

(define (transfer from-account to-account amount) ((from-account 'withdraw) amount) ((to-account 'deposit) amount))

Louis Reasoner afirma que aquí hay un problema, y que necesitamos usar un método más sofisticado, como el requerido para tratar con el problema del intercambio. ¿Tiene razón Louis? Si no, ¿cuál es la diferencia esencial entre el problema de la transferencia y el problema del intercambio? (Debe asumir que el saldo en `from/account` es al menos `amount`.)

Ejercicio 3.45: Louis Reasoner piensa que nuestro sistema de cuentas bancarias es innecesariamente complejo y propenso a errores ahora que los depósitos y retiros no se serializan automáticamente. Sugiere que `make/account/and/serializer` debería haber exportado el serializador (para su uso por procedimientos como `serialized/exchange`) además de (en lugar de en lugar de) usarlo para serializar cuentas y depósitos como lo hacía `make/account`. Propone redefinir las cuentas de la siguiente manera:

(define (make-account-and-serializer balance) (define (withdraw amount) (if (>= balance amount) (begin (set! balance (- balance amount)) balance) "Fondos insuficientes")) (define (deposit amount) (set! balance (+ balance amount)) balance) (let ((balance-serializer (make-serializer))) (define (dispatch m) (cond ((eq? m 'withdraw) (balance-serializer withdraw)) ((eq? m 'deposit) (balance-serializer deposit)) ((eq? m 'balance) balance) ((eq? m 'serializer) balance-serializer) (else (error "Solicitud desconocida: MAKE-ACCOUNT" m)))) dispatch))

Luego, los depósitos se manejan como con el `make/account` original:

(define (deposit account amount) ((account 'deposit) amount))

Explica qué está mal con el razonamiento de Louis. En particular, considera lo que sucede cuando se llama a `serialized/exchange`.

Implementando serializadores

Implementamos serializadores en términos de un mecanismo de sincronización más primitivo llamado *mutex*. Un mutex es un objeto que admite dos operaciones: el mutex puede ser *adquirido*, y el mutex puede ser *liberado*. Una vez que un mutex ha sido adquirido, ninguna otra operación de adquisición en ese mutex puede proceder hasta que el mutex sea liberado.[44] En nuestra implementación, cada serializador tiene un mutex asociado. Dado un procedimiento `p`, el serializador devuelve un procedimiento que adquiere el mutex, ejecuta `p`, y luego libera el mutex. Esto asegura que solo uno de los procedimientos producidos por el serializador pueda estar ejecutándose a la vez, que es precisamente la propiedad de serialización que necesitamos garantizar.

(define (make-serializer) (let ((mutex (make-mutex))) (lambda (p) (define (serialized-p . args) (mutex 'acquire) (let ((val (apply p args))) (mutex 'release) val)) serialized-p)))

El mutex es un objeto mutable (aquí usaremos una lista de un elemento, a la que nos referiremos como una *celda*) que puede contener el valor verdadero o falso. Cuando el valor es falso, el mutex está disponible para ser adquirido. Cuando el valor es verdadero, el mutex no está disponible, y cualquier proceso que intente adquirir el mutex debe esperar.

Nuestro constructor de mutex `make/mutex` comienza inicializando el contenido de la celda en falso. Para adquirir el mutex, probamos la celda. Si el mutex está disponible, establecemos el contenido de la celda en verdadero y procedemos. De lo contrario, esperamos en un bucle, intentando adquirir una y otra vez, hasta que encontramos que el mutex está disponible.[45] Para liberar el mutex, establecemos el contenido de la celda en falso.

(define (make-mutex) (let ((cell (list false))) (define (the-mutex m) (cond ((eq? m 'acquire) (if (test-and-set! cell) (the-mutex 'acquire)) ; reintentar ((eq? m 'release) (clear! cell)))) the-mutex)) (define (clear! cell) (set-car! cell false))

`test/and/set!` prueba la celda y devuelve el resultado de la prueba. Además, si la prueba fue falsa, `test/and/set!` establece el contenido de la celda en verdadero antes de devolver falso. Podemos expresar este comportamiento

como el siguiente procedimiento:

```
(define (test-and-set! cell) (if (car cell) true (begin (set-car! cell true) false)))
```

Sin embargo, esta implementación de `test/and/set!` no es suficiente tal como está. Hay una sutileza crucial aquí, que es el lugar esencial donde el control de la concurrencia entra en el sistema: la operación `test/and/set!` debe realizarse *atómicamente*. Es decir, debemos garantizar que, una vez que un proceso ha probado la celda y la ha encontrado falsa, el contenido de la celda será realmente establecido en verdadero antes de que cualquier otro proceso pueda probar la celda. Si no hacemos esta garantía, entonces el mutex puede fallar de una manera similar a la falla de la cuenta bancaria en la Figura 3.29. (Ver Ejercicio 3.46.)

La implementación real de `test/and/set!` depende de los detalles de cómo nuestro sistema ejecuta los procesos concurrentes. Por ejemplo, podríamos estar ejecutando procesos concurrentes en un procesador secuencial usando un mecanismo de división de tiempo que cicla a través de los procesos, permitiendo que cada proceso se ejecute durante un corto tiempo antes de interrumpirlo y pasar al siguiente proceso. En ese caso, `test/and/set!` puede funcionar deshabilitando la división de tiempo durante la prueba y el establecimiento.[46] Alternativamente, las computadoras de multiprocesamiento proporcionan instrucciones que soportan operaciones atómicas directamente en hardware.[47]

Ejercicio 3.46: Supongamos que implementamos `test/and/set!` usando un procedimiento ordinario como se muestra en el texto, sin intentar hacer la operación atómica. Dibuja un diagrama de tiempo como el de la Figura 3.29 para demostrar cómo la implementación del mutex puede fallar al permitir que dos procesos adquieran el mutex al mismo tiempo.

Ejercicio 3.47: Un semáforo (de tamaño *n*) es una generalización de un mutex. Al igual que un mutex, un semáforo admite operaciones de adquisición y liberación, pero es más general en el sentido de que hasta *n* procesos pueden adquirirlo concurrentemente. Los procesos adicionales que intentan adquirir el semáforo deben esperar a las operaciones de liberación. Dé implementaciones de semáforos

1. en términos de mutexes
2. en términos de operaciones atómicas `test/and/set!`.

Bloqueo mutuo

Ahora que hemos visto cómo implementar serializadores, podemos ver que el intercambio de cuentas todavía tiene un problema, incluso con el procedimiento `serialized/exchange` anterior. Imagina que Pedro intenta intercambiar $a1$ con $a2$ mientras Pablo intenta intercambiar concurrentemente $a2$ con $a1$. Supongamos que el proceso de Pedro llega al punto en que ha entrado en un procedimiento serializado que protege a $a1$ y, justo después, el proceso de Pablo entra en un procedimiento serializado que protege a $a2$. Ahora Pedro no puede proceder (a entrar en un procedimiento serializado que protege a $a2$) hasta que Pablo salga del procedimiento serializado que protege a $a2$. De manera similar, Pablo no puede proceder hasta que Pedro salga del procedimiento serializado que protege a $a1$. Cada proceso se detiene para siempre, esperando al otro. Esta situación se llama un *bloqueo mutuo*. El bloqueo mutuo siempre es un peligro en los sistemas que proporcionan acceso concurrente a múltiples recursos compartidos.

Una forma de evitar el bloqueo mutuo en esta situación es dar a cada cuenta un número de identificación único y reescribir `serialized/exchange` para que un proceso siempre intente entrar primero en un procedimiento que proteja la cuenta de menor número. Aunque este método funciona bien para el problema del intercambio, hay otras situaciones que requieren técnicas de evitación de bloqueo mutuo más sofisticadas, o donde el bloqueo mutuo no puede evitarse en absoluto. (Ver Ejercicio 3.48 y Ejercicio 3.49.)[48]

> **Ejercicio 3.48:** Explique en detalle por qué el método de evitación de bloqueo mutuo descrito anteriormente, (es decir, las cuentas están numeradas, y cada proceso intenta adquirir primero la cuenta con el número más pequeño) evita el bloqueo mutuo en el problema del intercambio. Vuelva a escribir `serialized/exchange` para incorporar esta idea. (También necesitará modificar `make/account` para que cada cuenta se cree con un número, al que se puede acceder enviando un mensaje apropiado.)
>
> **Ejercicio 3.49:** Dé un escenario donde el mecanismo de evitación de interbloqueo descrito anteriormente no funcione. (Pista: En el problema del intercambio, cada proceso sabe de antemano a qué cuentas necesitará tener acceso. Considere una situación en la que un proceso debe tener acceso a algunos recursos compartidos antes de que pueda saber qué recursos compartidos adicionales requerirá.)

Concurrencia, tiempo y comunicación

Hemos visto cómo la programación de sistemas concurrentes requiere controlar el orden de los eventos cuando diferentes procesos acceden al estado compartido, y hemos visto cómo lograr este control mediante el uso

juicioso de serializadores. Pero los problemas de la concurrencia son más profundos, porque, desde un punto de vista fundamental, no siempre está claro qué se entiende por "estado compartido".

Mecanismos como `test/and/set!` requieren que los procesos examinen una bandera compartida global en momentos arbitrarios. Esto es problemático e ineficiente de implementar en los procesadores modernos de alta velocidad, donde debido a técnicas de optimización como el pipelining y la memoria caché, el contenido de la memoria puede no estar en un estado consistente en todo momento. En los sistemas de multiprocesamiento contemporáneos, por lo tanto, el paradigma del serializador está siendo suplantado por nuevos enfoques para el control de la concurrencia.[49]

Los aspectos problemáticos del estado compartido también surgen en sistemas grandes y distribuidos. Por ejemplo, imagine un sistema bancario distribuido donde las sucursales bancarias individuales mantienen valores locales para los saldos bancarios y los comparan periódicamente con los valores mantenidos por otras sucursales. En tal sistema, el valor del "saldo de la cuenta" estaría indeterminado, excepto justo después de la sincronización. Si Pedro deposita dinero en una cuenta que comparte con Pablo, ¿cuándo deberíamos decir que el saldo de la cuenta ha cambiado—cuando cambia el saldo en la sucursal local, o no hasta después de la sincronización? Y si Pablo accede a la cuenta desde una sucursal diferente, ¿cuáles son las restricciones razonables que se deben imponer al sistema bancario para que el comportamiento sea "correcto"? Lo único que podría importar para la corrección es el comportamiento observado por Pedro y Pablo individualmente y el "estado" de la cuenta inmediatamente después de la sincronización. Las preguntas sobre el saldo "real" de la cuenta o el orden de los eventos entre sincronizaciones pueden ser irrelevantes o sin significado.[50]

El fenómeno básico aquí es que la sincronización de diferentes procesos, el establecimiento de un estado compartido o la imposición de un orden a los eventos requiere comunicación entre los procesos. En esencia, cualquier noción de tiempo en el control de la concurrencia debe estar íntimamente ligada a la comunicación.[51] Es intrigante que una conexión similar entre el tiempo y la comunicación también surja en la Teoría de la Relatividad, donde la velocidad de la luz (la señal más rápida que se puede usar para sincronizar eventos) es una constante fundamental que relaciona el tiempo y el espacio. Las complejidades que encontramos al tratar con el tiempo y el estado en nuestros modelos computacionales pueden, de hecho, reflejar una complejidad fundamental del universo físico.

Flujos

Hemos adquirido una buena comprensión de la asignación como herramienta en el modelado, así como una apreciación de los complejos problemas que plantea la asignación. Es hora de preguntarnos si podríamos haber hecho las cosas de otra manera, para evitar algunos de estos problemas. En esta sección, exploramos un enfoque alternativo para modelar el estado, basado en estructuras de datos llamadas *flujos*. Como veremos, los flujos pueden mitigar parte de la complejidad del modelado del estado.

Demos un paso atrás y revisemos de dónde proviene esta complejidad. En un intento de modelar fenómenos del mundo real, tomamos algunas decisiones aparentemente razonables: modelamos objetos del mundo real con estado local mediante objetos computacionales con variables locales. Identificamos la variación temporal en el mundo real con la variación temporal en la computadora. Implementamos la variación temporal de los estados de los objetos del modelo en la computadora con asignaciones a las variables locales de los objetos del modelo.

¿Hay otro enfoque? ¿Podemos evitar identificar el tiempo en la computadora con el tiempo en el mundo modelado? ¿Debemos hacer que el modelo cambie con el tiempo para modelar fenómenos en un mundo cambiante? Pensemos en el tema en términos de funciones matemáticas. Podemos describir el comportamiento variable en el tiempo de una cantidad x como una función del tiempo $x(t)$. Si nos concentramos en x instante por instante, pensamos en ella como una cantidad cambiante. Sin embargo, si nos concentramos en toda la historia temporal de los valores, no enfatizamos el cambio—la función en sí no cambia.[52]

Si el tiempo se mide en pasos discretos, entonces podemos modelar una función de tiempo como una secuencia (posiblemente infinita). En esta sección, veremos cómo modelar el cambio en términos de secuencias que representan las historias temporales de los sistemas que se modelan. Para lograr esto, introducimos nuevas estructuras de datos llamadas *flujos*. Desde un punto de vista abstracto, un flujo es simplemente una secuencia. Sin embargo, descubriremos que la implementación directa de flujos como listas (como en Section 2.2.1) no revela completamente el poder del procesamiento de flujos. Como alternativa, introducimos la técnica de *evaluación retardada*, que nos permite representar secuencias muy grandes (incluso infinitas) como flujos.

El procesamiento de flujos nos permite modelar sistemas que tienen estado sin usar nunca la asignación o datos mutables. Esto tiene implicaciones importantes, tanto teóricas como prácticas, porque podemos construir

modelos que evitan los inconvenientes inherentes a la introducción de la asignación. Por otro lado, el marco de los flujos plantea dificultades propias, y la cuestión de qué técnica de modelado conduce a sistemas más modulares y más fáciles de mantener sigue abierta.

Los Flujos son Listas Retardadas

Como vimos en Section 2.2.3, las secuencias pueden servir como interfaces estándar para combinar módulos de programas. Formulamos abstracciones poderosas para manipular secuencias, como `map`, `filter` y `accumulate`, que capturan una amplia variedad de operaciones de una manera que es a la vez sucinta y elegante.

Desafortunadamente, si representamos las secuencias como listas, esta elegancia se compra al precio de una severa ineficiencia con respecto tanto al tiempo como al espacio requeridos por nuestros cálculos. Cuando representamos manipulaciones sobre secuencias como transformaciones de listas, nuestros programas deben construir y copiar estructuras de datos (que pueden ser enormes) en cada paso de un proceso.

Para ver por qué esto es cierto, comparemos dos programas para calcular la suma de todos los números primos en un intervalo. El primer programa está escrito en estilo iterativo estándar:[53]

(define (sum-primes a b) (define (iter count accum) (cond ((> count b) accum) ((prime? count) (iter (+ count 1) (+ count accum))) (else (iter (+ count 1) accum)))) (iter a 0))

El segundo programa realiza el mismo cálculo usando las operaciones de secuencia de Section 2.2.3:

(define (sum-primes a b) (accumulate + 0 (filter prime? (enumerate-interval a b))))

Al realizar el cálculo, el primer programa solo necesita almacenar la suma que se está acumulando. En contraste, el filtro en el segundo programa no puede hacer ninguna prueba hasta que `enumerate/interval` haya construido una lista completa de los números en el intervalo. El filtro genera otra lista, que a su vez se pasa a `accumulate` antes de ser colapsada para formar una suma. Tal almacenamiento intermedio grande no es necesario para el primer programa, que podemos pensar como enumerando el intervalo incrementalmente, agregando cada primo a la suma a medida que se genera.

La ineficiencia en el uso de listas se hace dolorosamente evidente si usamos

el paradigma de secuencia para calcular el segundo primo en el intervalo de 10,000 a 1,000,000 evaluando la expresión

(car (cdr (filter prime? (enumerate-interval 10000 1000000))))

Esta expresión encuentra el segundo primo, pero el costo computacional es escandaloso. Construimos una lista de casi un millón de enteros, filtramos esta lista probando la primalidad de cada elemento, y luego ignoramos casi todo el resultado. En un estilo de programación más tradicional, entrelazaríamos la enumeración y el filtrado, y nos detendríamos cuando llegáramos al segundo primo.

Los flujos son una idea inteligente que permite usar manipulaciones de secuencias sin incurrir en los costos de manipular secuencias como listas. Con los flujos podemos lograr lo mejor de ambos mundos: podemos formular programas elegantemente como manipulaciones de secuencias, mientras alcanzamos la eficiencia del cálculo incremental. La idea básica es organizar la construcción de un flujo solo parcialmente, y pasar la construcción parcial al programa que consume el flujo. Si el consumidor intenta acceder a una parte del flujo que aún no ha sido construida, el flujo construirá automáticamente justo lo suficiente más de sí mismo para producir la parte requerida, preservando así la ilusión de que todo el flujo existe. En otras palabras, aunque escribiremos programas como si estuviéramos procesando secuencias completas, diseñamos nuestra implementación de flujos para entrelazar automática y transparentemente la construcción del flujo con su uso.

A primera vista, los flujos son solo listas con nombres diferentes para los procedimientos que los manipulan. Hay un constructor, `cons/stream`, y dos selectores, `stream/car` y `stream/cdr`, que satisfacen las restricciones

(stream-car (cons-stream x y)) = x (stream-cdr (cons-stream x y)) = y

Hay un objeto distinguible, `the/empty/stream`, que no puede ser el resultado de ninguna operación `cons/stream`, y que puede identificarse con el predicado `stream/null?`.[54] Así podemos hacer y usar flujos, de la misma manera que podemos hacer y usar listas, para representar datos agregados dispuestos en una secuencia. En particular, podemos construir análogos de flujo de las operaciones de lista del Capítulo 2, como `list/ref`, `map`, y `for/each`:[55]

(define (stream-ref s n) (if (= n 0) (stream-car s) (stream-ref (stream-cdr s) (- n 1)))) (define (stream-map proc s) (if (stream-null? s) the-empty-stream (cons-stream (proc (stream-car s)) (stream-map proc (stream-cdr s))))) (define (stream-for-each proc s) (if (stream-null? s) 'done (begin (proc

(stream-car s)) (stream-for-each proc (stream-cdr s)))))

`stream/for/each` es útil para ver flujos:

(define (display-stream s) (stream-for-each display-line s)) (define (display-line x) (newline) (display x))

Para hacer que la implementación del flujo entrelaze automática y transparentemente la construcción de un flujo con su uso, haremos arreglos para que el `cdr` de un flujo se evalúe cuando se acceda a él mediante el procedimiento `stream/cdr` en lugar de cuando se construya el flujo mediante `cons/stream`. Esta elección de implementación recuerda nuestra discusión sobre los números racionales en Section 2.1.2, donde vimos que podemos elegir implementar los números racionales de modo que la reducción del numerador y el denominador a su mínima expresión se realice ya sea en el momento de la construcción o en el momento de la selección. Las dos implementaciones de números racionales producen la misma abstracción de datos, pero la elección tiene un efecto en la eficiencia. Existe una relación similar entre los flujos y las listas ordinarias. Como abstracción de datos, los flujos son lo mismo que las listas. La diferencia es el momento en que se evalúan los elementos. Con las listas ordinarias, tanto el `car` como el `cdr` se evalúan en el momento de la construcción. Con los flujos, el `cdr` se evalúa en el momento de la selección.

Nuestra implementación de flujos se basará en una forma especial llamada `delay`. Evaluar `(delay` ⟨*exp*⟩`)` no evalúa la expresión ⟨*exp*$\kern0.08em \rangle$, sino que devuelve un llamado *objeto retrasado*, que podemos pensar como una "promesa" de evaluar ⟨*exp*$\kern0.08em\rangle$ en algún momento futuro. Como complemento a `delay`, hay un procedimiento llamado `force` que toma un objeto retrasado como argumento y realiza la evaluación—en efecto, obligando al `delay` a cumplir su promesa. Veremos a continuación cómo se pueden implementar `delay` y `force`, pero primero usemos estos para construir flujos.

`cons/stream` es una forma especial definida de modo que

(cons-stream $< a >$ $< b >$)

es equivalente a

(cons $< a >$ (delay $< b >$))

Lo que esto significa es que construiremos flujos usando pares. Sin embargo, en lugar de poner el valor del resto del flujo en el `cdr` del par, pondremos allí una promesa de calcular el resto si alguna vez se solicita. `stream/car` y

`stream/cdr` ahora se pueden definir como procedimientos:

(define (stream-car stream) (car stream)) (define (stream-cdr stream) (force (cdr stream)))

`stream/car` selecciona el `car` del par; `stream/cdr` selecciona el `cdr` del par y evalúa la expresión retardada que se encuentra allí para obtener el resto del flujo.[56]

La implementación del flujo en acción

Para ver cómo se comporta esta implementación, analicemos el cálculo de primos "escandaloso" que vimos anteriormente, reformulado en términos de flujos:

(stream-car (stream-cdr (stream-filter prime? (stream-enumerate-interval 10000 1000000))))

Veremos que, de hecho, funciona eficientemente.

Comenzamos llamando a `stream/enumerate/interval` con los argumentos 10,000 y 1,000,000. `stream/enumerate/interval` es el análogo de flujo de `enumerate/interval` (Section 2.2.3):

(define (stream-enumerate-interval low high) (if (> low high) the-empty-stream (cons-stream low (stream-enumerate-interval (+ low 1) high))))

y por lo tanto el resultado devuelto por `stream/enumerate/interval`, formado por el `cons/stream`, es[57]

(cons 10000 (delay (stream-enumerate-interval 10001 1000000)))

Es decir, `stream/enumerate/interval` devuelve un flujo representado como un par cuyo `car` es 10,000 y cuyo `cdr` es una promesa de enumerar más del intervalo si se solicita. Este flujo se filtra ahora para primos, usando el análogo de flujo del procedimiento `filter` (Section 2.2.3):

(define (stream-filter pred stream) (cond ((stream-null? stream) the-empty-stream) ((pred (stream-car stream)) (cons-stream (stream-car stream) (stream-filter pred (stream-cdr stream)))) (else (stream-filter pred (stream-cdr stream)))))

`stream/filter` prueba el `stream/car` del flujo (el `car` del par, que es 10,000). Como este no es primo, `stream/filter` examina el `stream/cdr` de su flujo de entrada. La llamada a `stream/cdr` fuerza la evaluación del

`stream/enumerate/interval` retardado, que ahora devuelve

(cons 10001 (delay (stream-enumerate-interval 10002 1000000)))

`stream/filter` ahora mira el `stream/car` de este flujo, 10,001, ve que este tampoco es primo, fuerza otro `stream/cdr`, y así sucesivamente, hasta que `stream/enumerate/interval` produce el primo 10,007, con lo cual `stream/filter`, según su definición, devuelve

(cons-stream (stream-car stream) (stream-filter pred (stream-cdr stream)))

que en este caso es

(cons 10007 (delay (stream-filter prime? (cons 10008 (delay (stream-enumerate-interval 10009 1000000))))))

Este resultado se pasa ahora a `stream/cdr` en nuestra expresión original. Esto fuerza el `stream/filter` retardado, que a su vez sigue forzando el `stream/enumerate/interval` retardado hasta que encuentra el siguiente primo, que es 10,009. Finalmente, el resultado pasado a `stream/car` en nuestra expresión original es

(cons 10009 (delay (stream-filter prime? (cons 10010 (delay (stream-enumerate-interval 10011 1000000))))))

`stream/car` devuelve 10,009, y el cálculo está completo. Solo se probaron tantos enteros para la primalidad como fueron necesarios para encontrar el segundo primo, y el intervalo se enumeró solo hasta donde fue necesario para alimentar el filtro de primos.

En general, podemos pensar en la evaluación retardada como programación "dirigida por la demanda", en la que cada etapa en el proceso de flujo se activa solo lo suficiente para satisfacer la siguiente etapa. Lo que hemos hecho es desacoplar el orden real de los eventos en el cálculo de la estructura aparente de nuestros procedimientos. Escribimos procedimientos como si los flujos existieran "todos a la vez" cuando, en realidad, el cálculo se realiza incrementalmente, como en los estilos de programación tradicionales.

Implementando `delay` y `force`

Aunque `delay` y `force` pueden parecer operaciones misteriosas, su implementación es en realidad bastante sencilla. `delay` debe empaquetar una expresión para que pueda evaluarse más tarde a demanda, y podemos lograr esto simplemente tratando la expresión como el cuerpo de un procedimiento. `delay` puede ser una forma especial tal que

(delay < *exp* >)

es azúcar sintáctico para

(lambda () < *exp* >)

`force` simplemente llama al procedimiento (sin argumentos) producido por `delay`, por lo que podemos implementar `force` como un procedimiento:

(define (force delayed-object) (delayed-object))

Esta implementación es suficiente para que `delay` y `force` funcionen como se anuncia, pero hay una optimización importante que podemos incluir. En muchas aplicaciones, terminamos forzando el mismo objeto retardado muchas veces. Esto puede llevar a una grave ineficiencia en los programas recursivos que involucran flujos. (Ver Ejercicio 3.57.) La solución es construir objetos retardados de modo que la primera vez que se fuerzan, almacenen el valor que se calcula. Las forzadas posteriores simplemente devolverán el valor almacenado sin repetir el cálculo. En otras palabras, implementamos `delay` como un procedimiento memoizado de propósito especial similar al descrito en el Ejercicio 3.27. Una forma de lograr esto es usar el siguiente procedimiento, que toma como argumento un procedimiento (sin argumentos) y devuelve una versión memoizada del procedimiento. La primera vez que se ejecuta el procedimiento memoizado, guarda el resultado calculado. En evaluaciones posteriores, simplemente devuelve el resultado.

(define (memo-proc proc) (let ((already-run? false) (result false)) (lambda () (if (not already-run?) (begin (set! result (proc)) (set! already-run? true) result) result))))

`delay` se define entonces de modo que `(delay` ⟨*exp*⟩`)` es equivalente a

(memo-proc (lambda () < *exp* >))

y `force` se define como anteriormente.[58]

Ejercicio 3.50: Completa la siguiente definición, que generaliza `stream/map` para permitir procedimientos que toman múltiples argumentos, análogos a `map` en Section 2.2.1, Nota 12.

(define (stream-map proc . argstreams) (if (< ?? > (car argstreams)) the-empty-stream (< ?? > (apply proc (map < ?? > argstreams)) (apply stream-map (cons proc (map < ?? > argstreams))))))

Ejercicio 3.51: Para echar un vistazo más de cerca a la evaluación

retardada, usaremos el siguiente procedimiento, que simplemente devuelve su argumento después de imprimirlo:

(define (show x) (display-line x) x)

¿Qué imprime el intérprete en respuesta a la evaluación de cada expresión en la siguiente secuencia?[59]

(define x (stream-map show (stream-enumerate-interval 0 10))) (stream-ref x 5) (stream-ref x 7)

Ejercicio 3.52: Considere la secuencia de expresiones

(define sum 0) (define (accum x) (set! sum (+ x sum)) sum) (define seq (stream-map accum (stream-enumerate-interval 1 20))) (define y (stream-filter even? seq)) (define z (stream-filter (lambda (x) (= (remainder x 5) 0)) seq)) (stream-ref y 7) (display-stream z)

¿Cuál es el valor de `sum` después de que se evalúa cada una de las expresiones anteriores? ¿Cuál es la respuesta impresa al evaluar las expresiones `stream/ref` y `display/stream`? ¿Diferirían estas respuestas si hubiéramos implementado `(delay` ⟨*exp*⟩`)` simplemente como `(lambda ()` ⟨*exp*⟩`)` sin usar la optimización proporcionada por el procedimiento `memo/proc`? Explique.

Flujos Infinitos

Hemos visto cómo soportar la ilusión de manipular flujos como entidades completas aunque, en realidad, solo calculamos la parte del flujo que necesitamos acceder. Podemos explotar esta técnica para representar secuencias eficientemente como flujos, incluso si las secuencias son muy largas. Lo que es más sorprendente, podemos usar flujos para representar secuencias que son infinitamente largas. Por ejemplo, considere la siguiente definición del flujo de enteros positivos:

(define (integers-starting-from n) (cons-stream n (integers-starting-from (+ n 1)))) (define integers (integers-starting-from 1))

Esto tiene sentido porque `integers` será un par cuyo `car` es 1 y cuyo `cdr` es una promesa de producir los enteros que comienzan con 2. Este es un flujo infinitamente largo, pero en un momento dado solo podemos examinar una porción finita de él. Por lo tanto, nuestros programas nunca sabrán que todo el flujo infinito no está allí.

Usando `integers` podemos definir otros flujos infinitos, como el flujo de enteros que no son divisibles por 7:

(define (divisible? x y) (= (remainder x y) 0)) (define no-sevens (stream-filter (lambda (x) (not (divisible? x 7))) integers))

Luego podemos encontrar enteros no divisibles por 7 simplemente accediendo a los elementos de este flujo:

(stream-ref no-sevens 100) *117*

En analogía con `integers`, podemos definir el flujo infinito de números de Fibonacci:

(define (fibgen a b) (cons-stream a (fibgen b (+ a b)))) (define fibs (fibgen 0 1))

`fibs` es un par cuyo `car` es 0 y cuyo `cdr` es una promesa de evaluar `(fibgen 1 1)`. Cuando evaluamos este `(fibgen 1 1)` retardado, producirá un par cuyo `car` es 1 y cuyo `cdr` es una promesa de evaluar `(fibgen 1 2)`, y así sucesivamente.

Para ver un flujo infinito más emocionante, podemos generalizar el ejemplo de `no/sevens` para construir el flujo infinito de números primos, usando un método conocido como la *criba de Eratóstenes*.[60] Comenzamos con los enteros que comienzan con 2, que es el primer primo. Para obtener el resto de los primos, comenzamos filtrando los múltiplos de 2 del resto de los enteros. Esto deja un flujo que comienza con 3, que es el siguiente primo. Ahora filtramos los múltiplos de 3 del resto de este flujo. Esto deja un flujo que comienza con 5, que es el siguiente primo, y así sucesivamente. En otras palabras, construimos los primos mediante un proceso de cribado, descrito de la siguiente manera: Para cribar un flujo `S`, forme un flujo cuyo primer elemento sea el primer elemento de `S` y el resto del cual se obtiene filtrando todos los múltiplos del primer elemento de `S` del resto de `S` y cribando el resultado. Este proceso se describe fácilmente en términos de operaciones de flujo:

(define (sieve stream) (cons-stream (stream-car stream) (sieve (stream-filter (lambda (x) (not (divisible? x (stream-car stream)))) (stream-cdr stream))))) (define primes (sieve (integers-starting-from 2)))

Ahora para encontrar un primo en particular solo necesitamos pedirlo:

(stream-ref primes 50) *233*

Es interesante contemplar el sistema de procesamiento de señales establecido por `sieve`, mostrado en el "diagrama de Henderson" en la Figura 3.31.[61] El flujo de entrada alimenta a un "des`cons`er" que separa el primer elemento del flujo del resto del flujo. El primer elemento se usa para construir un filtro de divisibilidad, a través del cual se pasa el resto, y la salida del filtro se alimenta a otra caja de criba. Luego, el primer elemento original se `cons` en la salida de la criba interna para formar el flujo de salida. Por lo tanto, no solo el flujo es infinito, sino que el procesador de señales también es infinito, porque la criba contiene una criba dentro de ella.

Figura 3.31: La criba de primos vista como un sistema de procesamiento de señales.

Definiendo flujos implícitamente

Los flujos `integers` y `fibs` anteriores se definieron especificando procedimientos "generadores" que calculan explícitamente los elementos del flujo uno por uno. Una forma alternativa de especificar flujos es aprovechar la evaluación retardada para definir flujos implícitamente. Por ejemplo, la siguiente expresión define el flujo `ones` como un flujo infinito de unos:

```
(define ones (cons-stream 1 ones))
```

Esto funciona de manera muy similar a la definición de un procedimiento recursivo: `ones` es un par cuyo `car` es 1 y cuyo `cdr` es una promesa de evaluar `ones`. Evaluar el `cdr` nos da de nuevo un 1 y una promesa de evaluar `ones`, y así sucesivamente.

Podemos hacer cosas más interesantes manipulando flujos con operaciones como `add/streams`, que produce la suma elemento a elemento de dos flujos dados:[62]

```
(define (add-streams s1 s2) (stream-map + s1 s2))
```

Ahora podemos definir los enteros de la siguiente manera:

```
(define integers (cons-stream 1 (add-streams ones integers)))
```

Esto define `integers` como un flujo cuyo primer elemento es 1 y el resto del cual es la suma de `ones` e `integers`. Así, el segundo elemento de `integers` es 1 más el primer elemento de `integers`, o 2; el tercer elemento de `integers` es 1 más el segundo elemento de `integers`, o 3; y así sucesivamente. Esta definición funciona porque, en cualquier punto, se ha

generado suficiente del flujo `integers` como para poder alimentarlo de nuevo en la definición para producir el siguiente entero.

Podemos definir los números de Fibonacci en el mismo estilo:

(define fibs (cons-stream 0 (cons-stream 1 (add-streams (stream-cdr fibs) fibs))))

Esta definición dice que `fibs` es un flujo que comienza con 0 y 1, de tal manera que el resto del flujo puede generarse sumando `fibs` a sí mismo desplazado en un lugar:

1 1 2 3 5 8 13 21 ... = `(stream/cdr fibs)` 0 1 1 2 3 5 8 13 ... = `fibs`
0 1 1 2 3 5 8 13 21 34 ... = `fibs`

`scale/stream` es otro procedimiento útil para formular tales definiciones de flujos. Este multiplica cada elemento de un flujo por una constante dada:

(define (scale-stream stream factor) (stream-map (lambda (x) (* x factor)) stream))

Por ejemplo,

(define double (cons-stream 1 (scale-stream double 2)))

produce el flujo de potencias de 2: 1, 2, 4, 8, 16, 32, ….

Una definición alternativa del flujo de primos puede darse comenzando con los enteros y filtrándolos probando su primalidad. Necesitaremos el primer primo, 2, para empezar:

(define primes (cons-stream 2 (stream-filter prime? (integers-starting-from 3))))

Esta definición no es tan sencilla como parece, porque probaremos si un número *n* es primo comprobando si *n* es divisible por un primo (no por cualquier entero) menor o igual que $\sqrt{n}$:

(define (prime? n) (define (iter ps) (cond ((> (square (stream-car ps)) n) true) ((divisible? n (stream-car ps)) false) (else (iter (stream-cdr ps))))) (iter primes))

Esta es una definición recursiva, ya que `primes` se define en términos del predicado `prime?`, que a su vez utiliza el flujo `primes`. La razón por la que este procedimiento funciona es que, en cualquier punto, se ha generado suficiente del flujo `primes` para probar la primalidad de los números que

necesitamos comprobar a continuación. Es decir, para cada *n* que probamos para la primalidad, o bien *n* no es primo (en cuyo caso hay un primo ya generado que lo divide) o *n* es primo (en cuyo caso hay un primo ya generado, es decir, un primo menor que *n*, que es mayor que $\sqrt{n}$).[63]

Ejercicio 3.53: Sin ejecutar el programa, describa los elementos del flujo definido por

(define s (cons-stream 1 (add-streams s s)))

Ejercicio 3.54: Defina un procedimiento `mul/streams`, análogo a `add/streams`, que produzca el producto elemento a elemento de sus dos flujos de entrada. Use esto junto con el flujo de `integers` para completar la siguiente definición del flujo cuyo elemento n^{th} (contando desde 0) es el factorial de $n + 1$:

(define factorials (cons-stream 1 (mul-streams ⟨ ?? ⟩ ⟨ ?? ⟩)))

Ejercicio 3.55: Defina un procedimiento `partial/sums` que tome como argumento un flujo *S* y devuelva el flujo cuyos elementos son S_0, $S_0 + S_1$, $S_0 + S_1 + S_2$, ... Por ejemplo, `(partial/sums integers)` debería ser el flujo 1, 3, 6, 10, 15, ...

Ejercicio 3.56: Un problema famoso, planteado por primera vez por R. Hamming, es enumerar, en orden ascendente y sin repeticiones, todos los enteros positivos sin factores primos que no sean 2, 3 o 5. Una forma obvia de hacer esto es simplemente probar cada entero a su vez para ver si tiene algún factor que no sea 2, 3 y 5. Pero esto es muy ineficiente, ya que, a medida que los enteros se hacen más grandes, cada vez menos de ellos cumplen el requisito. Como alternativa, llamemos al flujo de números requerido `S` y observemos los siguientes hechos sobre él.

- `S` comienza con 1.
- Los elementos de `(scale/stream S 2)` son también elementos de `S`.
- Lo mismo es cierto para `(scale/stream S 3)` y `(scale/stream 5 S)`.
- Estos son todos los elementos de `S`.

Ahora todo lo que tenemos que hacer es combinar elementos de estas fuentes. Para esto definimos un procedimiento `merge` que combina dos flujos ordenados en un flujo de resultado ordenado, eliminando repeticiones:

(define (merge s1 s2) (cond ((stream-null? s1) s2) ((stream-null? s2) s1) (else (let ((s1car (stream-car s1)) (s2car (stream-car s2))) (cond ((< s1car s2car) (cons-stream s1car (merge (stream-cdr s1) s2))) ((> s1car s2car) (cons-stream s2car (merge s1 (stream-cdr s2)))) (else (cons-stream s1car (merge (stream-cdr s1) (stream-cdr s2)))))))))

Entonces el flujo requerido puede construirse con `merge`, de la siguiente manera:

(define S (cons-stream 1 (merge $\langle$?? $\rangle$ $\langle$?? $\rangle$)))

Rellene las expresiones que faltan en los lugares marcados con ⟨??$ \kern0.08em\rangle$ arriba.

Ejercicio 3.57: ¿Cuántas sumas se realizan cuando calculamos el n^{th} número de Fibonacci usando la definición de `fibs` basada en el procedimiento `add/streams`? Muestre que el número de sumas sería exponencialmente mayor si hubiéramos implementado `(delay` ⟨*exp*⟩`)` simplemente como `(lambda ()` ⟨*exp*⟩`)`, sin usar la optimización proporcionada por el procedimiento `memo/proc` descrito en Section 3.5.1.[64]

Ejercicio 3.58: Dé una interpretación del flujo calculado por el siguiente procedimiento:

(define (expand num den radix) (cons-stream (quotient (* num radix) den) (expand (remainder (* num radix) den) den radix)))

(`quotient` es una primitiva que devuelve el cociente entero de dos enteros.) ¿Cuáles son los elementos sucesivos producidos por `(expand 1 7 10)`? ¿Qué se produce por `(expand 3 8 10)`?

Ejercicio 3.59: En Section 2.5.3 vimos cómo implementar un sistema de aritmética polinómica representando los polinomios como listas de términos. De manera similar, podemos trabajar con *series de potencias*, como

$$e\^x = 1 + x + \displaystyle\frac{x\^2}{2} + \displaystyle\frac{x\^3}{3 \cdot 2} + \displaystyle\frac{x\^4}{4 \cdot 3 \cdot 2} + \dots,$$

$$\cos x = 1 - \displaystyle\frac{x\^2}{2} + \displaystyle\frac{x\^4}{4 \cdot 3 \cdot 2} - \dots,$$

$$\sin x = x - \displaystyle\frac{x\^3}{3 \cdot 2} + \displaystyle\frac{x\^5}{5 \cdot 4 \cdot 3 \cdot 2} - \dots$$

representadas como flujos infinitos. Representaremos la serie $a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots$ como el flujo cuyos elementos son los coeficientes $a_0, a_1, a_2, a_3, \ldots$.

1. La integral de la serie $a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots$ es la serie

   $$c + a_0 x + {1\over2} a_1 x^2 + {1\over3} a_2 x^3 + {1\over4} a_3 x^4 + \dots,$$

   donde c es cualquier constante. Defina un procedimiento `integrate/series` que tome como entrada un flujo $a_0, a_1, a_2, \ldots$ que representa una serie de potencias y devuelva el flujo a_0, ${1\over2}a_1$, ${1\over3}a_2$, … de coeficientes de los términos no constantes de la integral de la serie. (Dado que el resultado no tiene término constante, no representa una serie de potencias; cuando usemos `integrate/series`, haremos `cons` de la constante apropiada.)

2. La función $x \mapsto e^x$ es su propia derivada. Esto implica que e^x y la integral de e^x son la misma serie, excepto por el término constante, que es $e^0 = 1$. En consecuencia, podemos generar la serie para e^x como

   (define exp-series (cons-stream 1 (integrate-series exp-series)))

   Muestra cómo generar las series para seno y coseno, partiendo de los hechos de que la derivada del seno es el coseno y la derivada del coseno es el negativo del seno:

   (define cosine-series (cons-stream 1 < ?? >)) (define sine-series (cons-stream 0 < ?? >))

Ejercicio 3.60: Con las series de potencias representadas como flujos de coeficientes como en el Ejercicio 3.59, la suma de series se implementa mediante `add/streams`. Complete la definición del siguiente procedimiento para multiplicar series:

(define (mul-series s1 s2) (cons-stream < ?? > (add-streams < ?? > < ?? >)))

Puedes probar tu procedimiento verificando que $\sin^2 x + \cos^2 x = 1$, usando las series del Ejercicio 3.59.

Ejercicio 3.61: Sea S una serie de potencias (Ejercicio 3.59) cuyo término constante es 1. Supongamos que queremos encontrar la serie de potencias $1/S$, es decir, la serie X tal que $SX = 1$. Escribimos $S = 1 + SR$

donde S_R es la parte de S después del término constante. Entonces podemos resolver para X de la siguiente manera:

$$\begin{array}{r@{{}={}}l} S \cdot X & 1, \\ (1 + S_R) \cdot X & 1, \\ X + S_R \cdot X & 1, \\ X & 1 - S_R \cdot X. \end{array}$$

En otras palabras, X es la serie de potencias cuyo término constante es 1 y cuyos términos de orden superior vienen dados por el negativo de S_R por X. Usa esta idea para escribir un procedimiento `invert/unit/series` que calcule $1/S$ para una serie de potencias S con término constante 1. Necesitarás usar `mul/series` del Ejercicio 3.60.

Ejercicio 3.62: Use los resultados del Ejercicio 3.60 y del Ejercicio 3.61 para definir un procedimiento `div/series` que divida dos series de potencias. `div/series` debería funcionar para cualquier par de series, siempre que la serie del denominador comience con un término constante no nulo. (Si el denominador tiene un término constante cero, entonces `div/series` debería señalar un error.) Muestre cómo usar `div/series` junto con el resultado del Ejercicio 3.59 para generar la serie de potencias para la tangente.

Explotando el Paradigma de los Flujos

Los flujos con evaluación retardada pueden ser una poderosa herramienta de modelado, proporcionando muchos de los beneficios del estado local y la asignación. Además, evitan algunos de los enredos teóricos que acompañan a la introducción de la asignación en un lenguaje de programación.

El enfoque de flujos puede ser esclarecedor porque nos permite construir sistemas con diferentes límites de módulo que los sistemas organizados en torno a la asignación a variables de estado. Por ejemplo, podemos pensar en una serie de tiempo completa (o señal) como un foco de interés, en lugar de los valores de las variables de estado en momentos individuales. Esto hace que sea conveniente combinar y comparar componentes del estado de diferentes momentos.

Formulando iteraciones como procesos de flujo

En Section 1.2.1, introdujimos los procesos iterativos, que proceden actualizando las variables de estado. Ahora sabemos que podemos representar el estado como un flujo "atemporal" de valores en lugar de como un conjunto de variables a actualizar. Adoptemos esta perspectiva al revisar el procedimiento de la raíz cuadrada de Section 1.1.7. Recordemos que la

idea es generar una secuencia de conjeturas cada vez mejores para la raíz cuadrada de *x* aplicando una y otra vez el procedimiento que mejora las conjeturas:

(define (sqrt-improve guess x) (average guess (/ x guess)))

En nuestro procedimiento `sqrt` original, hicimos que estas conjeturas fueran los valores sucesivos de una variable de estado. En su lugar, podemos generar el flujo infinito de conjeturas, comenzando con una conjetura inicial de 1:[65]

(define (sqrt-stream x) (define guesses (cons-stream 1.0 (stream-map (lambda (guess) (sqrt-improve guess x)) guesses))) guesses)

(display-stream (sqrt-stream 2)) *1. 1.5 1.4166666666666665 1.4142156862745097 1.4142135623746899* ...

Podemos generar más y más términos del flujo para obtener conjeturas cada vez mejores. Si queremos, podemos escribir un procedimiento que siga generando términos hasta que la respuesta sea lo suficientemente buena. (Ver Ejercicio 3.64.)

Otra iteración que podemos tratar de la misma manera es generar una aproximación a π, basada en la serie alternante que vimos en la Sección 1.3.1:

$${\pi\over4} = 1 - {1\over3} + {1\over5} - {1\over7} + \dots.$$

Primero generamos el flujo de los sumandos de la serie (los recíprocos de los enteros impares, con signos alternos). Luego tomamos el flujo de las sumas de más y más términos (usando el procedimiento `partial/sums` del Ejercicio 3.55) y escalamos el resultado por 4:

(define (pi-summands n) (cons-stream (/ 1.0 n) (stream-map - (pi-summands (+ n 2))))) (define pi-stream (scale-stream (partial-sums (pi-summands 1)) 4))

(display-stream pi-stream) *4. 2.666666666666667 3.466666666666667 2.8952380952380956 3.3396825396825403 2.9760461760461765 3.2837384837384844 3.017071817071818* ...

Esto nos da un flujo de aproximaciones cada vez mejores a π, aunque las aproximaciones convergen bastante lentamente. Ocho términos de la secuencia acotan el valor de π entre 3.284 y 3.017.

Hasta ahora, nuestro uso del enfoque de flujo de estados no es muy diferente

de la actualización de variables de estado. Pero los flujos nos dan la oportunidad de hacer algunos trucos interesantes. Por ejemplo, podemos transformar un flujo con un *acelerador de secuencia* que convierte una secuencia de aproximaciones en una nueva secuencia que converge al mismo valor que la original, solo que más rápido.

Un tal acelerador, debido al matemático suizo del siglo XVIII Leonhard Euler, funciona bien con secuencias que son sumas parciales de series alternas (series de términos con signos alternos). En la técnica de Euler, si S_n es el n^{th} término de la secuencia de suma original, entonces la secuencia acelerada tiene términos

$$S_{n+1} - {(S_{n+1} - S_n)^2 \over S_{n-1} - 2S_n + S_{n+1}}\,.$$

Así, si la secuencia original se representa como un flujo de valores, la secuencia transformada viene dada por

```
(define (euler-transform s)
  (let ((s0 (stream-ref s 0))           ; S_{n-1}
        (s1 (stream-ref s 1))           ; S_n
        (s2 (stream-ref s 2)))          ; S_{n+1}
    (cons-stream (- s2 (/ (square (- s2 s1))
                          (+ s0 (* -2 s1) s2)))
                 (euler-transform (stream-cdr s)))))
```

Podemos demostrar la aceleración de Euler con nuestra secuencia de aproximaciones a π:

```
(display-stream (euler-transform pi-stream))
```

3.166666666666667
3.1333333333333337 *3.1452380952380956* *3.13968253968254*
3.1427128427128435 *3.1408813408813416* *3.142071817071818*
3.1412548236077655 …

Aún mejor, podemos acelerar la secuencia acelerada, y recursivamente acelerar esa, y así sucesivamente. A saber, creamos un flujo de flujos (una estructura que llamaremos *tableau*) en la que cada flujo es la transformada del anterior:

```
(define (make-tableau transform s)
  (cons-stream s (make-tableau transform (transform s))))
```

El tableau tiene la forma

$$\vbox{ \offinterlineskip \halign{ \strut \hfil \ #\ \hfil & \hfil \ #\ \hfil & \hfil \ #\ \hfil & \hfil \ #\ \hfil & \hfil \ #\ \hfil & \hfil \ #\ \hfil \cr $ s_{00} $ & $ s_{01} $ & $ s_{02} $ & $ s_{03} $ & $ s_{04} $ & $ \dots $ \cr & $ s_{10} $ & $ s_{11} $ & $ s_{12} $ & $ s_{13} $ & $ \dots $ \cr & & $ s_{20} $ & $ s_{21} $ & $ s_{22} $ & $ \dots $ \cr & & & $ \dots $ & & \cr } }$$

Finalmente, formamos una secuencia tomando el primer término de cada fila del tableau:

(define (accelerated-sequence transform s) (stream-map stream-car (make-tableau transform s)))

Podemos demostrar este tipo de "super-aceleración" de la secuencia de π:

(display-stream (accelerated-sequence euler-transform pi-stream)) *4. 3.166666666666667 3.142105263157895 3.141599357319005 3.1415927140337785 3.1415926539752927 3.1415926535911765 3.141592653589778 ...*

El resultado es impresionante. Tomar ocho términos de la secuencia produce el valor correcto de π con 14 decimales. Si hubiéramos usado solo la secuencia original de π, necesitaríamos calcular del orden de 10^{13} términos (es decir, expandir la serie lo suficiente para que los términos individuales sean menores que 10^{-13}) para obtener esa precisión.

Podríamos haber implementado estas técnicas de aceleración sin usar flujos. Pero la formulación de flujos es particularmente elegante y conveniente porque toda la secuencia de estados está disponible para nosotros como una estructura de datos que puede ser manipulada con un conjunto uniforme de operaciones.

Ejercicio 3.63: Louis Reasoner pregunta por qué el procedimiento `sqrt/stream` no fue escrito de la siguiente manera más directa, sin la variable local `guesses`:

(define (sqrt-stream x) (cons-stream 1.0 (stream-map (lambda (guess) (sqrt-improve guess x)) (sqrt-stream x))))

Alyssa P. Hacker responde que esta versión del procedimiento es considerablemente menos eficiente porque realiza cálculos redundantes. Explique la respuesta de Alyssa. ¿Diferirían todavía las dos versiones en eficiencia si nuestra implementación de `delay` usara solo `(lambda ()` ⟨*exp*⟩`)` sin usar la optimización proporcionada por `memo/proc` (Section 3.5.1)?

Ejercicio 3.64: Escriba un procedimiento `stream/limit` que tome como argumentos un flujo y un número (la tolerancia). Debería examinar el flujo hasta encontrar dos elementos sucesivos que difieran en valor absoluto en menos que la tolerancia, y devolver el segundo de los dos elementos. Usando esto, podríamos calcular raíces cuadradas hasta una tolerancia dada mediante

(define (sqrt x tolerance) (stream-limit (sqrt-stream x) tolerance))

Ejercicio 3.65: Use la serie

$$\ln 2 = 1 - {1\over2} + {1\over3} - {1\over4} + \dots$$

para calcular tres secuencias de aproximaciones al logaritmo natural de 2, de la misma manera que lo hicimos anteriormente para π. ¿Con qué rapidez convergen estas secuencias?

Flujos infinitos de pares

En Section 2.2.3, vimos cómo el paradigma de la secuencia maneja los bucles anidados tradicionales como procesos definidos en secuencias de pares. Si generalizamos esta técnica a flujos infinitos, entonces podemos escribir programas que no se representan fácilmente como bucles, porque el "bucle" debe abarcar un conjunto infinito.

Por ejemplo, supongamos que queremos generalizar el procedimiento `prime/sum/pairs` de Section 2.2.3 para producir el flujo de pares de *todos* los enteros (i,j) con $i \leq j$ tal que $i + j$ es primo. Si `int/pairs` es la secuencia de todos los pares de enteros (i,j) con $i \leq j$, entonces nuestro flujo requerido es simplemente[66]

(stream-filter (lambda (pair) (prime? (+ (car pair) (cadr pair)))) int-pairs)

Nuestro problema, entonces, es producir el flujo `int/pairs`. Más generalmente, supongamos que tenemos dos flujos $S = (Si)$ y $T = (Tj)$, e imaginemos la matriz rectangular infinita

$$\vbox{ \offinterlineskip \halign{ \strut \hfil \ #\ \hfil & \hfil \ #\ \hfil & \hfil \ #\ \hfil & \hfil \ #\ \hfil \cr $ (S_0, T_0) $ & $ (S_0, T_1) $ & $ (S_0, T_2) $ & $ \dots $ \cr $ (S_1, T_0) $ & $ (S_1, T_1) $ & $ (S_1, T_2) $ & $ \dots $ \cr $ (S_2, T_0) $ & $ (S_2, T_1) $ & $ (S_2, T_2) $ & $ \dots $ \cr $ \dots $ & & & \cr } }$$

Deseamos generar un flujo que contenga todos los pares de la matriz que se encuentren en o por encima de la diagonal, es decir, los pares

$$\vbox{ \offinterlineskip \halign{ \strut \hfil \ #\ \hfil & \hfil \ #\ \hfil & \hfil \ #\ \hfil & \hfil \ #\ \hfil \cr $ (S_0, T_0) $ & $ (S_0, T_1) $ & $ (S_0, T_2) $ & $ \dots $ \cr & $ (S_1, T_1) $ & $ (S_1, T_2) $ & $ \dots $ \cr & & $ (S_2, T_2) $ & $ \dots $ \cr & & & $ \dots $ \cr } }$$

(Si tomamos tanto S como T como el flujo de enteros, entonces este será

nuestro flujo deseado `int/pairs`.)

Llamemos al flujo general de pares `(pairs S T)`, y considerémoslo compuesto de tres partes: el par (S_0, T_0), el resto de los pares de la primera fila, y los pares restantes:[67]

$$\vbox{ \offinterlineskip \halign{ \strut \hfil \ #\ \hfil & \vrule \hfil \ #\ \hfil & \hfil \ #\ \hfil & \hfil \ #\ \hfil \cr $ (S_0, T_0) $ & $ (S_0, T_1) $ & $ (S_0, T_2) $ & $ \dots $ \cr \noalign{\hrule} & $ (S_1, T_1) $ & $ (S_1, T_2) $ & $ \dots $ \cr & & $ (S_2, T_2) $ & $ \dots $ \cr & & & $ \dots $ \cr } }$$

Observe que la tercera pieza de esta descomposición (pares que no están en la primera fila) es (recursivamente) los pares formados a partir de `(stream/cdr S)` y `(stream/cdr T)`. También nótese que la segunda pieza (el resto de la primera fila) es

```
(stream-map (lambda (x) (list (stream-car s) x)) (stream-cdr t))
```

Por lo tanto, podemos formar nuestro flujo de pares de la siguiente manera:

```
(define (pairs s t) (cons-stream (list (stream-car s) (stream-car t))
( < combinar-de-alguna-manera > (stream-map (lambda (x) (list (stream-car
s) x)) (stream-cdr t)) (pairs (stream-cdr s) (stream-cdr t)))))
```

Para completar el procedimiento, debemos elegir alguna forma de combinar los dos flujos internos. Una idea es usar el análogo de flujo del procedimiento `append` de Section 2.2.1:

```
(define (stream-append s1 s2) (if (stream-null? s1) s2 (cons-stream (stream-
car s1) (stream-append (stream-cdr s1) s2))))
```

Sin embargo, esto no es adecuado para flujos infinitos, ya que toma todos los elementos del primer flujo antes de incorporar el segundo flujo. En particular, si intentamos generar todos los pares de enteros positivos usando

```
(pairs integers integers)
```

nuestro flujo de resultados intentará primero recorrer todos los pares con el primer entero igual a 1 y, por lo tanto, nunca producirá pares con ningún otro valor del primer entero.

Para manejar flujos infinitos, necesitamos idear un orden de combinación que asegure que cada elemento será alcanzado eventualmente si dejamos que nuestro programa se ejecute el tiempo suficiente. Una forma elegante de lograr esto es con el siguiente procedimiento `interleave`:[68]

(define (interleave s1 s2) (if (stream-null? s1) s2 (cons-stream (stream-car s1) (interleave s2 (stream-cdr s1)))))

Dado que `interleave` toma elementos alternadamente de los dos flujos, cada elemento del segundo flujo eventualmente encontrará su camino en el flujo intercalado, incluso si el primer flujo es infinito.

Así podemos generar el flujo de pares requerido como

(define (pairs s t) (cons-stream (list (stream-car s) (stream-car t)) (interleave (stream-map (lambda (x) (list (stream-car s) x)) (stream-cdr t)) (pairs (stream-cdr s) (stream-cdr t)))))

Ejercicio 3.66: Examine el flujo `(pairs integers integers)`. ¿Puede hacer algún comentario general sobre el orden en el que se colocan los pares en el flujo? Por ejemplo, ¿aproximadamente cuántos pares preceden al par (1, 100)? ¿el par (99, 100)? ¿el par (100, 100)? (Si puede hacer declaraciones matemáticas precisas aquí, tanto mejor. Pero siéntase libre de dar respuestas más cualitativas si se encuentra atascado.)

Ejercicio 3.67: Modifique el procedimiento `pairs` para que `(pairs integers integers)` produzca el flujo de *todos* los pares de enteros (i,j) (sin la condición $i \leq j$). Pista: Necesitará mezclar un flujo adicional.

Ejercicio 3.68: Louis Reasoner piensa que construir un flujo de pares a partir de tres partes es innecesariamente complicado. En lugar de separar el par (S_0, T_0) del resto de los pares de la primera fila, propone trabajar con toda la primera fila, de la siguiente manera:

(define (pairs s t) (interleave (stream-map (lambda (x) (list (stream-car s) x)) t) (pairs (stream-cdr s) (stream-cdr t))))

¿Funciona esto? Considere lo que sucede si evaluamos `(pairs integers integers)` usando la definición de `pairs` de Louis.

Ejercicio 3.69: Escriba un procedimiento `triples` que tome tres flujos infinitos, S, T, y U, y produzca el flujo de ternas (Si, Tj, Uk) tales que $i \leq j \leq k$. Use `triples` para generar el flujo de todas las ternas pitagóricas de enteros positivos, es decir, las ternas (i,j,k) tales que $i \leq j$ y $i^2 + j^2 = k^2$.

Ejercicio 3.70: Sería bueno poder generar flujos en los que los pares aparezcan en algún orden útil, en lugar de en el orden que resulta de un proceso de entrelazado *ad hoc*. Podemos usar una técnica similar al procedimiento `merge` del Ejercicio 3.56, si definimos una forma de decir

que un par de enteros es "menor que" otro. Una forma de hacerlo es definir una "función de ponderación" $W(i,j)$ y estipular que (i_1,j_1) es menor que (i_2,j_2) si $W(i_1,j_1) < W(i_2,j_2)$. Escriba un procedimiento `merge/weighted` que sea como `merge`, excepto que `merge/weighted` toma un argumento adicional `weight`, que es un procedimiento que calcula el peso de un par, y se usa para determinar el orden en que los elementos deben aparecer en el flujo fusionado resultante.[69] Usando esto, generalice `pairs` a un procedimiento `weighted/pairs` que tome dos flujos, junto con un procedimiento que calcule una función de ponderación, y genere el flujo de pares, ordenados según el peso. Use su procedimiento para generar

1. el flujo de todos los pares de enteros positivos (i,j) con $i \leq j$ ordenados según la suma $i + j$,

2. el flujo de todos los pares de enteros positivos (i,j) con $i \leq j$, donde ni i ni j son divisibles por 2, 3 o 5, y los pares están ordenados según la suma $2i + 3j + 5ij$.

Ejercicio 3.71: Los números que pueden expresarse como la suma de dos cubos de más de una manera a veces se llaman *números de Ramanujan*, en honor al matemático Srinivasa Ramanujan.[70] Los flujos ordenados de pares proporcionan una solución elegante al problema de calcular estos números. Para encontrar un número que pueda escribirse como la suma de dos cubos de dos maneras diferentes, solo necesitamos generar el flujo de pares de enteros (i,j) ponderados según la suma $i^3 + j^3$ (ver Ejercicio 3.70), y luego buscar en el flujo dos pares consecutivos con el mismo peso. Escriba un procedimiento para generar los números de Ramanujan. El primer número de este tipo es 1.729. ¿Cuáles son los cinco siguientes?

Ejercicio 3.72: De manera similar al Ejercicio 3.71, genere un flujo de todos los números que pueden escribirse como la suma de dos cuadrados de tres maneras diferentes (mostrando cómo pueden escribirse así).

Flujos como señales

Comenzamos nuestra discusión sobre los flujos describiéndolos como análogos computacionales de las "señales" en los sistemas de procesamiento de señales. De hecho, podemos usar flujos para modelar sistemas de procesamiento de señales de una manera muy directa, representando los valores de una señal en intervalos de tiempo sucesivos como elementos consecutivos de un flujo. Por ejemplo, podemos implementar un *integrador* o *sumador* que, para un flujo de entrada $x = (x_i)$, un valor inicial C, y un pequeño incremento dt, acumula la suma

$$S_i = C + \sum_{j=1}^i x_{\kern-0.07em j} \kern0.1em dt$$

y devuelve el flujo de valores $S = (S_i)$. El siguiente procedimiento `integral` recuerda la definición de "estilo implícito" del flujo de enteros (Section 3.5.2):

(define (integral integrand initial-value dt) (define int (cons-stream initial-value (add-streams (scale-stream integrand dt) int))) int)

La Figura 3.32 es una imagen de un sistema de procesamiento de señales que corresponde al procedimiento `integral`. El flujo de entrada se escala por *dt* y se pasa a través de un sumador, cuya salida se pasa de nuevo a través del mismo sumador. La autorreferencia en la definición de `int` se refleja en la figura por el bucle de retroalimentación que conecta la salida del sumador a una de las entradas.

Figura 3.32: El procedimiento `integral` visto como un sistema de procesamiento de señales.

Ejercicio 3.73: Podemos modelar circuitos eléctricos usando flujos para representar los valores de las corrientes o voltajes en una secuencia de tiempos. Por ejemplo, supongamos que tenemos un *circuito RC* que consiste en una resistencia de resistencia R y un capacitor de capacitancia C en serie. La respuesta de voltaje v del circuito a una corriente inyectada i está determinada por la fórmula de la Figura 3.33, cuya estructura se muestra en el diagrama de flujo de señales adjunto.

Figura 3.33: Un circuito RC y el diagrama de flujo de señales asociado.

Escriba un procedimiento `RC` que modele este circuito. `RC` debería tomar como entradas los valores de R, C y dt y debería devolver un procedimiento que tome como entradas un flujo que represente la corriente i y un valor inicial para el voltaje del capacitor v_0 y produzca como salida el flujo de voltajes v. Por ejemplo, debería poder usar `RC` para modelar un circuito RC con $R = 5$ ohmios, $C = 1$ faradio y un paso de tiempo de 0.5 segundos evaluando `(define RC1 (RC 5 1 0.5))`. Esto define `RC1` como un procedimiento que toma un flujo que representa la secuencia temporal de corrientes y un voltaje inicial del capacitor y produce el flujo de salida de voltajes.

Ejercicio 3.74: Alyssa P. Hacker está diseñando un sistema para procesar señales procedentes de sensores físicos. Una característica importante que desea producir es una señal que describa los *cruces por cero* de la señal de entrada. Es decir, la señal resultante debería ser $+1$ cada vez que la señal de entrada cambie de negativa a positiva, -1 cada vez que la señal de entrada cambie de positiva a negativa, y 0 en caso contrario. (Asuma que el signo de una entrada 0 es positivo.) Por ejemplo, una señal de entrada típica con su señal de cruce por cero asociada sería

… 1 2 1.5 1 0.5 -0.1 -2 -3 -2 -0.5 0.2 3 4 … … 0 0 0 0 0 -1 0 0 0 0 1 0 0 …

En el sistema de Alyssa, la señal del sensor se representa como un flujo `sense/data` y el flujo `zero/crossings` es el flujo correspondiente de cruces por cero. Alyssa primero escribe un procedimiento `sign/change/detector` que toma dos valores como argumentos y compara los signos de los valores para producir un 0, 1 o -1 apropiado. Luego construye su flujo de cruces por cero de la siguiente manera:

(define (make-zero-crossings input-stream last-value) (cons-stream (sign-change-detector (stream-car input-stream) last-value) (make-zero-crossings (stream-cdr input-stream) (stream-car input-stream)))) (define zero-crossings (make-zero-crossings sense-data 0))

La jefa de Alyssa, Eva Lu Ator, pasa y sugiere que este programa es aproximadamente equivalente al siguiente, que utiliza la versión generalizada de `stream/map` del Ejercicio 3.50:

(define zero-crossings (stream-map sign-change-detector sense-data ⟨ *expresión* ⟩))

Completa el programa proporcionando la ⟨*expresión*⟩ indicada.

Ejercicio 3.75: Desafortunadamente, el detector de cruce por cero de Alyssa en el Ejercicio 3.74 resulta ser insuficiente, porque la señal ruidosa del sensor conduce a cruces por cero espurios. Lem E. Tweakit, un especialista en hardware, sugiere que Alyssa suavice la señal para filtrar el ruido antes de extraer los cruces por cero. Alyssa sigue su consejo y decide extraer los cruces por cero de la señal construida promediando cada valor de los datos del sensor con el valor anterior. Ella explica el problema a su asistente, Louis Reasoner, quien intenta implementar la idea, alterando el programa de Alyssa de la siguiente manera:

(define (make-zero-crossings input-stream last-value) (let ((avpt (/ (+ (stream-car input-stream) last-value) 2))) (cons-stream (sign-change-

detector avpt last-value) (make-zero-crossings (stream-cdr input-stream) avpt))))

Esto no implementa correctamente el plan de Alyssa. Encuentra el error que Louis ha instalado y corrígelo sin cambiar la estructura del programa. (Pista: Necesitarás aumentar el número de argumentos de `make/zero/crossings`.)

Ejercicio 3.76: Eva Lu Ator tiene una crítica al enfoque de Louis en el Ejercicio 3.75. El programa que escribió no es modular, porque entremezcla la operación de suavizado con la extracción del cruce por cero. Por ejemplo, el extractor no debería tener que cambiarse si Alyssa encuentra una mejor manera de acondicionar su señal de entrada. Ayuda a Louis escribiendo un procedimiento `smooth` que tome un flujo como entrada y produzca un flujo en el que cada elemento sea el promedio de dos elementos sucesivos del flujo de entrada. Luego usa `smooth` como un componente para implementar el detector de cruce por cero de una manera más modular.

Flujos y Evaluación Retardada

El procedimiento `integral` al final de la sección anterior muestra cómo podemos usar flujos para modelar sistemas de procesamiento de señales que contienen bucles de retroalimentación. El bucle de retroalimentación para el sumador mostrado en la Figura 3.32 se modela por el hecho de que el flujo interno de `integral`, `int`, se define en términos de sí mismo:

(define int (cons-stream initial-value (add-streams (scale-stream integrand dt) int)))

La capacidad del intérprete para tratar con tal definición implícita depende del `delay` que se incorpora en `cons/stream`. Sin este `delay`, el intérprete no podría construir `int` antes de evaluar ambos argumentos de `cons/stream`, lo que requeriría que `int` ya estuviera definido. En general, `delay` es crucial para usar flujos para modelar sistemas de procesamiento de señales que contienen bucles. Sin `delay`, nuestros modelos tendrían que ser formulados de modo que las entradas a cualquier componente de procesamiento de señales serían evaluadas completamente antes de que la salida pudiera ser producida. Esto prohibiría los bucles.

Figura 3.34: Un "circuito de computadora analógica" que resuelve la ecuación $dy/dt = f(y)$.

Desafortunadamente, los modelos de flujo de sistemas con bucles pueden requerir usos de `delay` más allá del `delay` "oculto" proporcionado por `cons/stream`. Por ejemplo, la Figura 3.34 muestra un sistema de procesamiento de señales para resolver la ecuación diferencial $dy/dt = f(y)$ donde f es una función dada. La figura muestra un componente de mapeo, que aplica f a su señal de entrada, conectado en un bucle de retroalimentación a un integrador de una manera muy similar a la de los circuitos de computadora analógica que se usan realmente para resolver tales ecuaciones.

Asumiendo que se nos da un valor inicial y_0 para y, podríamos intentar modelar este sistema usando el procedimiento

(define (solve f y0 dt) (define y (integral dy y0 dt)) (define dy (stream-map f y)) y)

Este procedimiento no funciona, porque en la primera línea de `solve` la llamada a `integral` requiere que la entrada `dy` esté definida, lo cual no sucede hasta la segunda línea de `solve`.

Por otro lado, la intención de nuestra definición tiene sentido, porque podemos, en principio, comenzar a generar el flujo `y` sin conocer `dy`. De hecho, `integral` y muchas otras operaciones de flujo tienen propiedades similares a las de `cons/stream`, en el sentido de que podemos generar parte de la respuesta dada solo información parcial sobre los argumentos. Para `integral`, el primer elemento del flujo de salida es el `valor/inicial` especificado. Por lo tanto, podemos generar el primer elemento del flujo de salida sin evaluar el integrando `dy`. Una vez que conocemos el primer elemento de `y`, el `stream/map` en la segunda línea de `solve` puede comenzar a trabajar para generar el primer elemento de `dy`, que producirá el siguiente elemento de `y`, y así sucesivamente.

Para aprovechar esta idea, redefiniremos `integral` para que espere que el flujo del integrando sea un *argumento retardado*. `integral forzará` la evaluación del integrando solo cuando sea necesario para generar más que el primer elemento del flujo de salida:

(define (integral delayed-integrand initial-value dt) (define int (cons-stream initial-value (let ((integrand (force delayed-integrand))) (add-streams (scale-stream integrand dt) int)))) int)

Ahora podemos implementar nuestro procedimiento `solve` retrasando la evaluación de `dy` en la definición de `y`:[71]

(define (solve f y0 dt) (define y (integral (delay dy) y0 dt)) (define dy (stream-map f y)) y)

En general, cada llamador de `integral` debe ahora `retrasar` el argumento del integrando. Podemos demostrar que el procedimiento `solve` funciona aproximando $e \approx 2.718$ calculando el valor en $y = 1$ de la solución a la ecuación diferencial $dy/dt = y$ con la condición inicial $y(0) = 1$:

(stream-ref (solve (lambda (y) y) 1 0.001) 1000) *2.716924*

Ejercicio 3.77: El procedimiento `integral` usado anteriormente era análogo a la definición "implícita" del flujo infinito de enteros en Section 3.5.2. Alternativamente, podemos dar una definición de `integral` que es más parecida a `integers/starting/from` (también en Section 3.5.2):

(define (integral integrand initial-value dt) (cons-stream initial-value (if (stream-null? integrand) the-empty-stream (integral (stream-cdr integrand) (+ (* dt (stream-car integrand)) initial-value) dt))))

Cuando se usa en sistemas con bucles, este procedimiento tiene el mismo problema que nuestra versión original de `integral`. Modifique el procedimiento para que espere el `integrando` como un argumento retardado y, por lo tanto, pueda usarse en el procedimiento `solve` mostrado anteriormente.

Figura 3.35: Diagrama de flujo de señales para la solución de una ecuación diferencial lineal de segundo orden.

Ejercicio 3.78: Considere el problema de diseñar un sistema de procesamiento de señales para estudiar la ecuación diferencial lineal homogénea de segundo orden

$${d^2\!y \over dt^2} - a {dy \over dt} - by = 0.$$

El flujo de salida, que modela a y, es generado por una red que contiene un bucle. Esto se debe a que el valor de d^2y/dt^2 depende de los valores de y y dy/dt y ambos se determinan integrando d^2y/dt^2. El diagrama que nos gustaría codificar se muestra en la Figura 3.35. Escriba un procedimiento `solve/2nd` que tome como argumentos las constantes a, b y dt y los valores iniciales y_0 y dy_0 para y y dy/dt y genere el flujo de valores sucesivos de y.

Ejercicio 3.79: Generalice el procedimiento `solve/2nd` del Ejercicio 3.78 para que pueda usarse para resolver ecuaciones diferenciales generales de segundo orden $d^2y/dt^2 = f(dy/dt, y)$.

Figura 3.36: Un circuito RLC en serie.

Ejercicio 3.80: Un *circuito RLC en serie* consiste en una resistencia, un capacitor y un inductor conectados en serie, como se muestra en la Figura 3.36. Si R, L y C son la resistencia, inductancia y capacitancia, entonces las relaciones entre el voltaje (v) y la corriente (i) para los tres componentes se describen mediante las ecuaciones

$$v_R = i_R R, \qquad v_L = L \frac{di_L}{dt}, \qquad i_C = C \frac{dv_C}{dt},$$

y las conexiones del circuito dictan las relaciones

$$i_R = i_L = -i_C, \qquad v_C = v_L + v_R.$$

La combinación de estas ecuaciones muestra que el estado del circuito (resumido por v_C, el voltaje a través del capacitor, e i_L, la corriente en el inductor) se describe por el par de ecuaciones diferenciales

$$\frac{dv_C}{dt} = -\frac{i_L}{C}, \qquad \frac{di_L}{dt} = \frac{1}{L} v_C - \frac{R}{L} i_L.$$

El diagrama de flujo de señales que representa este sistema de ecuaciones diferenciales se muestra en la Figura 3.37.

Figura 3.37: Un diagrama de flujo de señales para la solución de un circuito RLC en serie.

Escriba un procedimiento `RLC` que tome como argumentos los parámetros R, L, y C del circuito y el incremento de tiempo dt. De manera similar al procedimiento `RC` del Ejercicio 3.73, `RLC` debería producir un procedimiento que tome los valores iniciales de las variables de estado, v_{C_0} e i_{L_0}, y produzca un par (usando `cons`) de los flujos de estados v_C e i_L. Usando `RLC`, genere el par de flujos que modela el comportamiento de un circuito RLC en serie con $R = 1$ ohmio, $C = 0.2$ faradios, $L = 1$ henrio, $dt = 0.1$ segundos, y valores iniciales $i_{L_0} = 0$ amperios y $v_{C_0} = 10$ voltios.

Evaluación de orden normal

Los ejemplos de esta sección ilustran cómo el uso explícito de `delay` y `force`

proporciona una gran flexibilidad de programación, pero los mismos ejemplos también muestran cómo esto puede hacer que nuestros programas sean más complejos. Nuestro nuevo procedimiento `integral`, por ejemplo, nos da el poder de modelar sistemas con bucles, pero ahora debemos recordar que `integral` debe llamarse con un integrando retardado, y cada procedimiento que use `integral` debe ser consciente de esto. En efecto, hemos creado dos clases de procedimientos: procedimientos ordinarios y procedimientos que toman argumentos retardados. En general, crear clases separadas de procedimientos nos obliga a crear también clases separadas de procedimientos de orden superior.[72]

Una forma de evitar la necesidad de dos clases diferentes de procedimientos es hacer que todos los procedimientos tomen argumentos retardados. Podríamos adoptar un modelo de evaluación en el que todos los argumentos de los procedimientos se retrasan automáticamente y los argumentos se fuerzan solo cuando son realmente necesarios (por ejemplo, cuando son requeridos por una operación primitiva). Esto transformaría nuestro lenguaje para usar la evaluación de orden normal, que describimos por primera vez cuando introdujimos el modelo de sustitución para la evaluación en Section 1.1.5. Convertir a la evaluación de orden normal proporciona una forma uniforme y elegante de simplificar el uso de la evaluación retardada, y esta sería una estrategia natural a adoptar si estuviéramos preocupados solo por el procesamiento de flujos. En Section 4.2, después de haber estudiado el evaluador, veremos cómo transformar nuestro lenguaje de esta manera. Desafortunadamente, incluir retardos en las llamadas a procedimientos causa estragos en nuestra capacidad para diseñar programas que dependen del orden de los eventos, como los programas que usan asignación, mutan datos o realizan entrada o salida. Incluso el único `delay` en `cons/stream` puede causar una gran confusión, como se ilustra en el Ejercicio 3.51 y el Ejercicio 3.52. Hasta donde se sabe, la mutabilidad y la evaluación retardada no se mezclan bien en los lenguajes de programación, y diseñar formas de lidiar con ambas a la vez es un área activa de investigación.

Modularidad de los Programas Funcionales y Modularidad de los Objetos

Como vimos en Section 3.1.2, uno de los principales beneficios de introducir la asignación es que podemos aumentar la modularidad de nuestros sistemas encapsulando, o "ocultando", partes del estado de un sistema grande dentro de variables locales. Los modelos de flujo pueden proporcionar una modularidad equivalente sin el uso de la asignación. Como ilustración, podemos reimplementar la estimación de Monte Carlo de π, que examinamos en Section 3.1.2, desde un punto de vista de procesamiento de flujos.

El problema clave de la modularidad era que deseábamos ocultar el estado interno de un generador de números aleatorios de los programas que usaban números aleatorios. Comenzamos con un procedimiento `rand/update`, cuyos valores sucesivos proporcionaban nuestro suministro de números aleatorios, y lo usamos para producir un generador de números aleatorios:

(define rand (let ((x random-init)) (lambda () (set! x (rand-update x)) x)))

En la formulación de flujos no hay un generador de números aleatorios *per se*, solo un flujo de números aleatorios producido por llamadas sucesivas a `rand/update`:

(define random-numbers (cons-stream random-init (stream-map rand-update random-numbers)))

Usamos esto para construir el flujo de resultados del experimento de Cesàro realizado en pares consecutivos en el flujo `random/numbers`:

(define cesaro-stream (map-successive-pairs (lambda (r1 r2) (= (gcd r1 r2) 1)) random-numbers)) (define (map-successive-pairs f s) (cons-stream (f (stream-car s) (stream-car (stream-cdr s))) (map-successive-pairs f (stream-cdr (stream-cdr s)))))

El `cesaro/stream` ahora se alimenta a un procedimiento `monte/carlo`, que produce un flujo de estimaciones de probabilidades. Los resultados se convierten luego en un flujo de estimaciones de π. Esta versión del programa no necesita un parámetro que indique cuántos ensayos realizar. Se obtienen mejores estimaciones de π (al realizar más experimentos) mirando más adentro en el flujo `pi`:

(define (monte-carlo experiment-stream passed failed) (define (next passed failed) (cons-stream (/ passed (+ passed failed)) (monte-carlo (stream-cdr experiment-stream) passed failed))) (if (stream-car experiment-stream) (next (+ passed 1) failed) (next passed (+ failed 1)))) (define pi (stream-map (lambda (p) (sqrt (/ 6 p))) (monte-carlo cesaro-stream 0 0)))

Hay una considerable modularidad en este enfoque, porque todavía podemos formular un procedimiento general `monte/carlo` que puede tratar con experimentos arbitrarios. Sin embargo, no hay asignación ni estado local.

> **Ejercicio 3.81:** El Ejercicio 3.6 discutió la generalización del generador de números aleatorios para permitir restablecer la secuencia de números aleatorios para producir secuencias repetibles de números "aleatorios". Produzca una formulación de flujo de este mismo generador que opere en un flujo de entrada de solicitudes para `generar` un nuevo número

aleatorio o para `restablecer` la secuencia a un valor especificado y que produzca el flujo deseado de números aleatorios. No use asignación en su solución.

Ejercicio 3.82: Rehaga el Ejercicio 3.5 sobre la integración de Monte Carlo en términos de flujos. La versión de flujo de `estimate/integral` no tendrá un argumento que indique cuántos ensayos realizar. En su lugar, producirá un flujo de estimaciones basadas en sucesivamente más ensayos.

Una visión del tiempo desde la programación funcional

Volvamos ahora a los problemas de los objetos y el estado que se plantearon al principio de este capítulo y examinémoslos bajo una nueva luz. Introdujimos la asignación y los objetos mutables para proporcionar un mecanismo para la construcción modular de programas que modelan sistemas con estado. Construimos objetos computacionales con variables de estado locales y usamos la asignación para modificar estas variables. Modelamos el comportamiento temporal de los objetos en el mundo por el comportamiento temporal de los objetos computacionales correspondientes.

Ahora hemos visto que los flujos proporcionan una forma alternativa de modelar objetos con estado local. Podemos modelar una cantidad cambiante, como el estado local de algún objeto, usando un flujo que representa la historia temporal de los estados sucesivos. En esencia, representamos el tiempo explícitamente, usando flujos, de modo que desacoplamos el tiempo en nuestro mundo simulado de la secuencia de eventos que tienen lugar durante la evaluación. De hecho, debido a la presencia de `delay`, puede haber poca relación entre el tiempo simulado en el modelo y el orden de los eventos durante la evaluación.

Para contrastar estos dos enfoques de modelado, reconsideremos la implementación de un "procesador de retiros" que monitorea el saldo en una cuenta bancaria. En Section 3.1.3 implementamos una versión simplificada de tal procesador:

(define (make-simplified-withdraw balance) (lambda (amount) (set! balance (- balance amount)) balance))

Las llamadas a `make/simplified/withdraw` producen objetos computacionales, cada uno con una variable de estado local `balance` que se decrementa con las sucesivas llamadas al objeto. El objeto toma una `cantidad` como argumento y devuelve el nuevo saldo. Podemos imaginar al usuario de una cuenta bancaria escribiendo una secuencia de entradas a tal

objeto y observando la secuencia de valores devueltos que se muestran en una pantalla.

Alternativamente, podemos modelar un procesador de retiros como un procedimiento que toma como entrada un saldo y un flujo de cantidades a retirar y produce el flujo de saldos sucesivos en la cuenta:

(define (stream-withdraw balance amount-stream) (cons-stream balance (stream-withdraw (- balance (stream-car amount-stream)) (stream-cdr amount-stream))))

`stream/withdraw` implementa una función matemática bien definida cuya salida está completamente determinada por su entrada. Supongamos, sin embargo, que la entrada `amount/stream` es el flujo de valores sucesivos escritos por el usuario y que el flujo de saldos resultante se muestra. Entonces, desde la perspectiva del usuario que está escribiendo valores y observando los resultados, el proceso de flujo tiene el mismo comportamiento que el objeto creado por `make/simplified/withdraw`. Sin embargo, con la versión de flujo, no hay asignación, no hay variable de estado local y, en consecuencia, ninguna de las dificultades teóricas que encontramos en Section 3.1.3. ¡Y sin embargo, el sistema tiene estado!

Esto es realmente notable. Aunque `stream/withdraw` implementa una función matemática bien definida cuyo comportamiento no cambia, la percepción del usuario aquí es la de interactuar con un sistema que tiene un estado cambiante. Una forma de resolver esta paradoja es darse cuenta de que es la existencia temporal del usuario la que impone el estado al sistema. Si el usuario pudiera dar un paso atrás de la interacción y pensar en términos de flujos de saldos en lugar de transacciones individuales, el sistema parecería sin estado.[73]

Desde el punto de vista de una parte de un proceso complejo, las otras partes parecen cambiar con el tiempo. Tienen un estado local oculto que varía en el tiempo. Si deseamos escribir programas que modelen este tipo de descomposición natural en nuestro mundo (tal como lo vemos desde nuestro punto de vista como parte de ese mundo) con estructuras en nuestra computadora, hacemos objetos computacionales que no son funcionales—deben cambiar con el tiempo. Modelamos el estado con variables de estado locales, y modelamos los cambios de estado con asignaciones a esas variables. Al hacer esto, hacemos que el tiempo de ejecución de un cálculo modele el tiempo en el mundo del que somos parte, y así obtenemos "objetos" en nuestra computadora.

Modelar con objetos es poderoso e intuitivo, en gran parte porque esto

coincide con la percepción de interactuar con un mundo del que somos parte. Sin embargo, como hemos visto repetidamente a lo largo de este capítulo, estos modelos plantean problemas espinosos de restringir el orden de los eventos y de sincronizar múltiples procesos. La posibilidad de evitar estos problemas ha estimulado el desarrollo de *lenguajes de programación funcionales*, que no incluyen ninguna provisión para la asignación o los datos mutables. En tal lenguaje, todos los procedimientos implementan funciones matemáticas bien definidas de sus argumentos, cuyo comportamiento no cambia. El enfoque funcional es extremadamente atractivo para tratar con sistemas concurrentes.[74]

Por otro lado, si miramos de cerca, podemos ver que los problemas relacionados con el tiempo también se infiltran en los modelos funcionales. Un área particularmente problemática surge cuando deseamos diseñar sistemas interactivos, especialmente aquellos que modelan interacciones entre entidades independientes. Por ejemplo, consideremos una vez más la implementación de un sistema bancario que permite cuentas bancarias conjuntas. En un sistema convencional que usa asignación y objetos, modelaríamos el hecho de que Pedro y Pablo comparten una cuenta haciendo que tanto Pedro como Pablo envíen sus solicitudes de transacción al mismo objeto de cuenta bancaria, como vimos en Section 3.1.3. Desde el punto de vista del flujo, donde no hay "objetos" *per se*, ya hemos indicado que una cuenta bancaria puede modelarse como un proceso que opera sobre un flujo de solicitudes de transacción para producir un flujo de respuestas. En consecuencia, podríamos modelar el hecho de que Pedro y Pablo tienen una cuenta bancaria conjunta fusionando el flujo de solicitudes de transacción de Pedro con el flujo de solicitudes de Pablo y alimentando el resultado al proceso de flujo de la cuenta bancaria, como se muestra en la Figura 3.38.

Figura 3.38: Una cuenta bancaria conjunta, modelada fusionando dos flujos de solicitudes de transacción.

El problema con esta formulación está en la noción de *fusión*. No servirá fusionar los dos flujos simplemente tomando alternativamente una solicitud de Pedro y una solicitud de Pablo. Supongamos que Pablo accede a la cuenta solo muy raramente. Difícilmente podríamos obligar a Pedro a esperar a que Pablo acceda a la cuenta antes de que pudiera emitir una segunda transacción. Como sea que se implemente tal fusión, debe entrelazar los dos flujos de transacciones de alguna manera que esté restringida por el "tiempo real" según lo perciben Pedro y Pablo, en el sentido de que, si Pedro y Pablo se encuentran, pueden acordar que ciertas transacciones se procesaron antes de la reunión, y otras transacciones se procesaron después de la reunión.[75]

Esta es precisamente la misma restricción con la que tuvimos que lidiar en Section 3.4.1, donde encontramos la necesidad de introducir sincronización explícita para asegurar un orden "correcto" de eventos en el procesamiento concurrente de objetos con estado. Así, en un intento de apoyar el estilo funcional, la necesidad de fusionar entradas de diferentes agentes reintroduce los mismos problemas que el estilo funcional pretendía eliminar.

Comenzamos este capítulo con el objetivo de construir modelos computacionales cuya estructura coincida con nuestra percepción del mundo real que estamos tratando de modelar. Podemos modelar el mundo como una colección de objetos separados, ligados al tiempo, que interactúan con el estado, o podemos modelar el mundo como una unidad única, atemporal y sin estado. Cada visión tiene poderosas ventajas, pero ninguna visión por sí sola es completamente satisfactoria. Aún no ha surgido una gran unificación.[76]

1. En realidad, esto no es del todo cierto. Una excepción fue el generador de números aleatorios en Section 1.2.6. Otra excepción involucró las tablas de operación/tipo que introdujimos en Section 2.4.3, donde los valores de dos llamadas a `get` con los mismos argumentos dependían de llamadas intermedias a `put`. Por otro lado, hasta que introducimos la asignación, no tenemos forma de crear tales procedimientos nosotros mismos.

2. El valor de una expresión `set!` depende de la implementación. `set!` debe usarse solo por su efecto, no por su valor.

El nombre `set!` refleja una convención de nomenclatura utilizada en Scheme: Las operaciones que cambian los valores de las variables (o que cambian las estructuras de datos, como veremos en Section 3.3) reciben nombres que terminan con un signo de exclamación. Esto es similar a la convención de designar predicados con nombres que terminan con un signo de interrogación.

3. Ya hemos usado `begin` implícitamente en nuestros programas, porque en Scheme el cuerpo de un procedimiento puede ser una secuencia de expresiones. Además, la parte ⟨*consecuente*$\kern0.06em\rangle$ de cada cláusula en una expresión `cond` puede ser una secuencia de expresiones en lugar de una sola expresión.

4. En la jerga de los lenguajes de programación, se dice que la variable `balance` está *encapsulada* dentro del procedimiento `new/withdraw`. La encapsulación refleja el principio general de diseño de sistemas conocido como el *principio de ocultación*: se puede hacer un sistema más modular y

robusto protegiendo partes del sistema de otras; es decir, proporcionando acceso a la información solo a aquellas partes del sistema que tienen "necesidad de saber".

5. En contraste con `new/withdraw` arriba, no tenemos que usar `let` para hacer que `balance` sea una variable local, ya que los parámetros formales ya son locales. Esto quedará más claro después de la discusión del modelo de entorno de evaluación en Section 3.2. (Ver también el Ejercicio 3.10.)

6. Una forma común de implementar `rand/update` es usar la regla de que x se actualiza a $ax + b$ módulo m, donde a, b y m son enteros elegidos apropiadamente. El Capítulo 3 de Knuth 1981 incluye una extensa discusión de técnicas para generar secuencias de números aleatorios y establecer sus propiedades estadísticas. Nótese que el procedimiento `rand/update` calcula una función matemática: Dada la misma entrada dos veces, produce la misma salida. Por lo tanto, la secuencia de números producida por `rand/update` ciertamente no es "aleatoria", si por "aleatorio" insistimos en que cada número en la secuencia no está relacionado con el número anterior. La relación entre la "aleatoriedad real" y las llamadas secuencias *pseudo-aleatorias*, que son producidas por cálculos bien determinados y sin embargo tienen propiedades estadísticas adecuadas, es una cuestión compleja que involucra difíciles problemas en matemáticas y filosofía. Kolmogorov, Solomonoff y Chaitin han hecho grandes progresos en aclarar estas cuestiones; se puede encontrar una discusión en Chaitin 1975.

7. Este teorema se debe a E. Cesàro. Véase la sección 4.5.2 de Knuth 1981 para una discusión y una prueba.

8. MIT Scheme proporciona tal procedimiento. Si a `random` se le da un entero exacto (como en Section 1.2.6) devuelve un entero exacto, pero si se le da un valor decimal (como en este ejercicio) devuelve un valor decimal.

9. No sustituimos la ocurrencia de `balance` en la expresión `set!` porque el ⟨*nombre*$\kern0.08em\rangle$ en un `set!` no se evalúa. Si lo sustituyéramos, obtendríamos `(set! 25 (- 25 amount))`, lo que no tiene sentido.

10. El fenómeno de que un solo objeto computacional sea accedido por más de un nombre se conoce como *aliasing*. La situación de la cuenta bancaria conjunta ilustra un ejemplo muy simple de un alias. En Section 3.3 veremos ejemplos mucho más complejos, como estructuras de datos compuestas "distintas" que comparten partes. Pueden ocurrir errores en nuestros programas si olvidamos que un cambio en un objeto también puede, como "efecto secundario", cambiar un objeto "diferente" porque los dos objetos "diferentes" son en realidad un solo objeto que aparece bajo diferentes alias.

Estos llamados *bugs de efectos secundarios* son tan difíciles de localizar y analizar que algunas personas han propuesto que los lenguajes de programación se diseñen de tal manera que no permitan efectos secundarios o aliasing (Lampson et al. 1981; Morris et al. 1980).

11. En vista de esto, es irónico que la programación introductoria se enseñe con mayor frecuencia en un estilo altamente imperativo. Esto puede ser un vestigio de una creencia, común durante las décadas de 1960 y 1970, de que los programas que llaman a procedimientos deben ser inherentemente menos eficientes que los programas que realizan asignaciones. (Steele 1977 refuta este argumento.) Alternativamente, puede reflejar la opinión de que la asignación paso a paso es más fácil de visualizar para los principiantes que la llamada a procedimientos. Cualquiera que sea la razón, a menudo carga a los programadores principiantes con preocupaciones del tipo "¿debería establecer esta variable antes o después de esa otra?" que pueden complicar la programación y oscurecer las ideas importantes.

12. La asignación introduce una sutileza en el paso 1 de la regla de evaluación. Como se muestra en el Ejercicio 3.8, la presencia de asignación nos permite escribir expresiones que producirán valores diferentes dependiendo del orden en que se evalúen las subexpresiones en una combinación. Por lo tanto, para ser precisos, deberíamos especificar un orden de evaluación en el paso 1 (por ejemplo, de izquierda a derecha o de derecha a izquierda). Sin embargo, este orden siempre debe considerarse un detalle de implementación, y uno nunca debe escribir programas que dependan de un orden en particular. Por ejemplo, un compilador sofisticado podría optimizar un programa variando el orden en que se evalúan las subexpresiones.

13. Si ya hay una ligadura para la variable en el marco actual, entonces se cambia la ligadura. Esto es conveniente porque permite la redefinición de símbolos; sin embargo, también significa que `define` puede usarse para cambiar valores, y esto plantea los problemas de la asignación sin usar explícitamente `set!`. Debido a esto, algunas personas prefieren que las redefiniciones de símbolos existentes señalen errores o advertencias.

14. El modelo de entorno no aclarará nuestra afirmación en Section 1.2.1 de que el intérprete puede ejecutar un procedimiento como `fact/iter` en una cantidad constante de espacio usando recursión de cola. Discutiremos la recursión de cola cuando tratemos con la estructura de control del intérprete en Section 5.4.

15. Si `W1` y `W2` comparten el mismo código físico almacenado en la computadora, o si cada uno guarda una copia del código, es un detalle de la

implementación. Para el intérprete que implementamos en el Capítulo 4, el código de hecho se comparte.

16. `set/car!` y `set/cdr!` devuelven valores dependientes de la implementación. Al igual que `set!`, deben usarse solo por su efecto.

17. Vemos de esto que las operaciones de mutación en listas pueden crear "basura" que no es parte de ninguna estructura accesible. Veremos en Section 5.3.2 que los sistemas de gestión de memoria de Lisp incluyen un *recolector de basura*, que identifica y recicla el espacio de memoria utilizado por pares innecesarios.

18. `get/new/pair` es una de las operaciones que debe implementarse como parte de la gestión de memoria requerida por una implementación de Lisp. Discutiremos esto en Section 5.3.1.

19. Los dos pares son distintos porque cada llamada a `cons` devuelve un nuevo par. Los símbolos son compartidos; en Scheme hay un único símbolo con un nombre dado. Dado que Scheme no proporciona ninguna forma de mutar un símbolo, esta compartición es indetectable. Nótese también que la compartición es lo que nos permite comparar símbolos usando `eq?`, que simplemente comprueba la igualdad de punteros.

20. Las sutilezas de tratar con la compartición de objetos de datos mutables reflejan los problemas subyacentes de "identidad" y "cambio" que se plantearon en Section 3.1.3. Mencionamos allí que admitir el cambio en nuestro lenguaje requiere que un objeto compuesto deba tener una "identidad" que sea algo diferente de las piezas de las que está compuesto. En Lisp, consideramos que esta "identidad" es la cualidad que se prueba con `eq?`, es decir, por igualdad de punteros. Dado que en la mayoría de las implementaciones de Lisp un puntero es esencialmente una dirección de memoria, estamos "resolviendo el problema" de definir la identidad de los objetos estipulando que un objeto de datos "en sí mismo$\kern0.1em$" es la información almacenada en algún conjunto particular de ubicaciones de memoria en la computadora. Esto es suficiente para programas Lisp simples, pero difícilmente es una forma general de resolver el problema de la "identidad" en modelos computacionales.

21. Por otro lado, desde el punto de vista de la implementación, la asignación requiere que modifiquemos el entorno, que es en sí mismo una estructura de datos mutable. Por lo tanto, la asignación y la mutación son equipotentes: cada una puede implementarse en términos de la otra.

22. Si el primer elemento es el último elemento de la cola, el puntero frontal será la lista vacía después de la eliminación, lo que marcará la cola como

vacía; no necesitamos preocuparnos por actualizar el puntero trasero, que todavía apuntará al elemento eliminado, porque `empty/queue?` solo mira el puntero frontal.

23. Ten cuidado de no hacer que el intérprete intente imprimir una estructura que contenga ciclos. (Ver Ejercicio 3.13.)

24. Debido a que `assoc` usa `equal?`, puede reconocer claves que son símbolos, números o estructuras de lista.

25. Así, el primer par del esqueleto es el objeto que representa la tabla "en sí misma$\kern0.1em$"; es decir, un puntero a la tabla es un puntero a este par. Este mismo par del esqueleto siempre inicia la tabla. Si no hubiéramos organizado las cosas de esta manera, `insert!` tendría que devolver un nuevo valor para el inicio de la tabla cuando agregara un nuevo registro.

26. Un sumador completo es un elemento de circuito básico utilizado para sumar dos números binarios. Aquí A y B son los bits en las posiciones correspondientes en los dos números a sumar, y $\rm C_{in}$ es el bit de acarreo de la suma una posición a la derecha. El circuito genera SUM, que es el bit de suma en la posición correspondiente, y $\rm C_{out}$, que es el bit de acarreo que se propagará a la izquierda.

27. Estos procedimientos son simplemente azúcar sintáctico que nos permite usar la sintaxis procedimental ordinaria para acceder a los procedimientos locales de los objetos. Es sorprendente que podamos intercambiar el papel de "procedimientos" y "datos" de una manera tan simple. Por ejemplo, si escribimos `(wire 'get/signal)` pensamos en `wire` como un procedimiento que se llama con el mensaje `get/signal` como entrada. Alternativamente, escribir `(get/signal wire)` nos anima a pensar en `wire` como un objeto de datos que es la entrada a un procedimiento `get/signal`. La verdad del asunto es que, en un lenguaje en el que podemos tratar con procedimientos como objetos, no hay ninguna diferencia fundamental entre "procedimientos" y "datos", y podemos elegir nuestro azúcar sintáctico para permitirnos programar en el estilo que elijamos.

28. La agenda es una lista con encabezado, como las tablas en Section 3.3.3, pero como la lista está encabezada por el tiempo, no necesitamos un encabezado ficticio adicional (como el símbolo `*table*` utilizado con las tablas).

29. Observe que la expresión `if` en este procedimiento no tiene una expresión ⟨*alternativa*$\kern0.08em\rangle$. Tal "sentencia `if` de un solo brazo" se usa para decidir si hacer algo, en lugar de para seleccionar entre dos expresiones. Una expresión `if` devuelve un valor no especificado si el

predicado es falso y no hay ⟨*alternativa*$\kern0.08em\rangle$.

30. De esta manera, el tiempo actual siempre será el tiempo de la acción procesada más recientemente. Almacenar este tiempo al principio de la agenda asegura que seguirá estando disponible incluso si el segmento de tiempo asociado ha sido eliminado.

31. La propagación de restricciones apareció por primera vez en el increíblemente avanzado sistema SKETCHPAD de Ivan Sutherland (1963). Un hermoso sistema de propagación de restricciones basado en el lenguaje Smalltalk fue desarrollado por Alan Borning (1977) en el Centro de Investigación de Xerox Palo Alto. Sussman, Stallman y Steele aplicaron la propagación de restricciones al análisis de circuitos eléctricos (Sussman and Stallman 1975; Sussman and Steele 1980). TK!Solver (Konopasek and Jayaraman 1984) es un extenso entorno de modelado basado en restricciones.

32. El `setter` podría no ser una restricción. En nuestro ejemplo de temperatura, usamos `user` como el `setter`.

33. El formato orientado a expresiones es conveniente porque evita la necesidad de nombrar las expresiones intermedias en un cálculo. Nuestra formulación original del lenguaje de restricciones es engorrosa de la misma manera que muchos lenguajes son engorrosos al tratar con operaciones en datos compuestos. Por ejemplo, si quisiéramos calcular el producto $(a + b) \cdot (c + d)$, donde las variables representan vectores, podríamos trabajar en "estilo imperativo", usando procedimientos que establecen los valores de los argumentos de vectores designados pero que no devuelven ellos mismos vectores como valores:

(v-sum a b temp1) (v-sum c d temp2) (v-prod temp1 temp2 answer)

Alternativamente, podríamos tratar con expresiones, usando procedimientos que devuelven vectores como valores, y así evitar mencionar explícitamente `temp1` y `temp2`:

(define answer (v-prod (v-sum a b) (v-sum c d)))

Dado que Lisp nos permite devolver objetos compuestos como valores de procedimientos, podemos transformar nuestro lenguaje de restricciones de estilo imperativo en un estilo orientado a expresiones como se muestra en este ejercicio. En lenguajes que son pobres en el manejo de objetos compuestos, como Algol, Basic y Pascal (a menos que uno use explícitamente variables de puntero de Pascal), uno generalmente está atascado con el estilo imperativo al manipular objetos compuestos. Dada la ventaja del formato

orientado a expresiones, uno podría preguntarse si hay alguna razón para haber implementado el sistema en estilo imperativo, como lo hicimos en esta sección. Una razón es que el lenguaje de restricciones no orientado a expresiones proporciona un manejo de los objetos de restricción (por ejemplo, el valor del procedimiento `adder`) así como de los objetos conectores. Esto es útil si deseamos extender el sistema con nuevas operaciones que se comunican con las restricciones directamente en lugar de solo indirectamente a través de operaciones en conectores. Aunque es fácil implementar el estilo orientado a expresiones en términos de la implementación imperativa, es muy difícil hacer lo contrario.

34. La mayoría de los procesadores reales ejecutan en realidad unas pocas operaciones a la vez, siguiendo una estrategia llamada *pipelining*. Aunque esta técnica mejora enormemente la utilización efectiva del hardware, solo se usa para acelerar la ejecución de un flujo de instrucciones secuencial, mientras se conserva el comportamiento del programa secuencial.

35. Para citar un graffiti visto en una pared de un edificio de Cambridge: “El tiempo es un dispositivo que se inventó para evitar que todo sucediera a la vez.”

36. Un fallo aún peor para este sistema podría ocurrir si las dos operaciones `set!` intentaran cambiar el saldo simultáneamente, en cuyo caso los datos reales que aparecen en la memoria podrían terminar siendo una combinación aleatoria de la información que están escribiendo los dos procesos. La mayoría de las computadoras tienen interbloqueos en las operaciones primitivas de escritura en memoria, que protegen contra tal acceso simultáneo. Sin embargo, incluso este tipo de protección aparentemente simple plantea desafíos de implementación en el diseño de computadoras de multiprocesamiento, donde se requieren elaborados protocolos de *coherencia de caché* para asegurar que los diversos procesadores mantendrán una vista consistente del contenido de la memoria, a pesar del hecho de que los datos pueden ser replicados ("cacheados") entre los diferentes procesadores para aumentar la velocidad de acceso a la memoria.

37. El programa factorial en Section 3.1.3 ilustra esto para un solo proceso secuencial.

38. Las columnas muestran el contenido de la billetera de Peter, la cuenta conjunta (en el Banco1), la billetera de Paul y la cuenta privada de Paul (en el Banco2), antes y después de cada retiro (R) y depósito (D). Peter retira $10 del Banco1; Paul deposita $5 en el Banco2, luego retira $25 del Banco1.

39. Una forma más formal de expresar esta idea es decir que los programas

concurrentes son inherentemente *no deterministas*. Es decir, no se describen por funciones de un solo valor, sino por funciones cuyos resultados son conjuntos de valores posibles. En Section 4.3 estudiaremos un lenguaje para expresar cálculos no deterministas.

40. `parallel/execute` no es parte del estándar de Scheme, pero puede implementarse en MIT Scheme. En nuestra implementación, los nuevos procesos concurrentes también se ejecutan concurrentemente con el proceso original de Scheme. Además, en nuestra implementación, el valor devuelto por `parallel/execute` es un objeto de control especial que puede usarse para detener los procesos recién creados.

41. Hemos simplificado `exchange` aprovechando el hecho de que nuestro mensaje `deposit` acepta cantidades negativas. (¡Este es un error grave en nuestro sistema bancario!)

42. Si los saldos de las cuentas comienzan en $10, $20 y $30, entonces después de cualquier número de intercambios concurrentes, los saldos aún deberían ser $10, $20 y $30 en algún orden. Serializar los depósitos en cuentas individuales no es suficiente para garantizar esto. Ver Ejercicio 3.43.

43. Ejercicio 3.45 investiga por qué los depósitos y retiros ya no son serializados automáticamente por la cuenta.

44. El término "mutex" es una abreviatura de *exclusión mutua*. El problema general de organizar un mecanismo que permita a los procesos concurrentes compartir recursos de forma segura se llama el problema de exclusión mutua. Nuestro mutex es una variante simple del mecanismo de *semáforo* (ver Ejercicio 3.47), que se introdujo en el Sistema de Multiprogramación "THE" desarrollado en la Universidad Tecnológica de Eindhoven y nombrado por las iniciales de la universidad en holandés (Dijkstra 1968a). Las operaciones de adquisición y liberación se llamaron originalmente P y V, de las palabras holandesas *passeren* (pasar) y *vrijgeven* (liberar), en referencia a los semáforos utilizados en los sistemas ferroviarios. La exposición clásica de Dijkstra (Dijkstra 1968b) fue una de las primeras en presentar claramente los problemas del control de concurrencia, y mostró cómo usar semáforos para manejar una variedad de problemas de concurrencia.

45. En la mayoría de los sistemas operativos de tiempo compartido, los procesos que están bloqueados por un mutex no pierden el tiempo en una "espera ocupada" como la anterior. En su lugar, el sistema programa otro proceso para que se ejecute mientras el primero está esperando, y el proceso bloqueado se despierta cuando el mutex está disponible.

46. En MIT Scheme para un solo procesador, que utiliza un modelo de

división de tiempo, `test/and/set!` se puede implementar de la siguiente manera:

(define (test-and-set! cell) (without-interrupts (lambda () (if (car cell) true (begin (set-car! cell true) false)))))

`without/interrupts` deshabilita las interrupciones de división de tiempo mientras se ejecuta su argumento de procedimiento.

47. Existen muchas variantes de tales instrucciones, incluidas test-and-set, test-and-clear, swap, compare-and-exchange, load-reserve y store-conditional, cuyo diseño debe ser cuidadosamente adaptado a la interfaz procesador-memoria de la máquina. Un problema que surge aquí es determinar qué sucede si dos procesos intentan adquirir el mismo recurso exactamente al mismo tiempo usando tal instrucción. Esto requiere algún mecanismo para tomar una decisión sobre qué proceso obtiene el control. Tal mecanismo se llama un *árbitro*. Los árbitros generalmente se reducen a algún tipo de dispositivo de hardware. Desafortunadamente, es posible probar que no se puede construir físicamente un árbitro justo que funcione el 100% del tiempo a menos que se le permita al árbitro un tiempo arbitrariamente largo para tomar su decisión. El fenómeno fundamental aquí fue observado originalmente por el filósofo francés del siglo XIV Jean Buridan en su comentario sobre el *De caelo* de Aristóteles. Buridan argumentó que un perro perfectamente racional colocado entre dos fuentes de alimento igualmente atractivas morirá de hambre, porque es incapaz de decidir a cuál ir primero.

48. La técnica general para evitar el bloqueo mutuo numerando los recursos compartidos y adquiriéndolos en orden se debe a Havender (1968). Las situaciones donde el bloqueo mutuo no puede evitarse requieren métodos de *recuperación de bloqueo mutuo*, que implican que los procesos "retrocedan" del estado de bloqueo mutuo y lo intenten de nuevo. Los mecanismos de recuperación de bloqueo mutuo se utilizan ampliamente en los sistemas de gestión de bases de datos, un tema que se trata en detalle en Gray and Reuter 1993.

49. Una de esas alternativas a la serialización se llama *sincronización de barrera*. El programador permite que los procesos concurrentes se ejecuten como deseen, pero establece ciertos puntos de sincronización ("barreras") a través de los cuales ningún proceso puede proceder hasta que todos los procesos hayan alcanzado la barrera. Los procesadores modernos proporcionan instrucciones de máquina que permiten a los programadores establecer puntos de sincronización en lugares donde se requiere consistencia. El PowerPC, por ejemplo, incluye para este propósito dos instrucciones llamadas SYNC y EIEIO (Ejecución forzada en orden de entrada/

salida).

50. Este puede parecer un punto de vista extraño, pero hay sistemas que funcionan de esta manera. Los cargos internacionales a las cuentas de tarjetas de crédito, por ejemplo, normalmente se liquidan por país, y los cargos realizados en diferentes países se concilian periódicamente. Por lo tanto, el saldo de la cuenta puede ser diferente en diferentes países.

51. Para los sistemas distribuidos, esta perspectiva fue seguida por Lamport (1978), quien mostró cómo usar la comunicación para establecer "relojes globales" que pueden usarse para establecer ordenamientos en los eventos en sistemas distribuidos.

52. Los físicos a veces adoptan esta visión al introducir las "líneas de mundo" de las partículas como un dispositivo para razonar sobre el movimiento. También hemos mencionado ya (Section 2.2.3) que esta es la forma natural de pensar sobre los sistemas de procesamiento de señales. Exploraremos aplicaciones de los flujos al procesamiento de señales en Section 3.5.3.

53. Supongamos que tenemos un predicado `prime?` (por ejemplo, como en Section 1.2.6) que prueba la primalidad.

54. En la implementación de MIT, `the/empty/stream` es lo mismo que la lista vacía `'()`, y `stream/null?` es lo mismo que `null?`.

55. Esto debería molestarte. El hecho de que estemos definiendo procedimientos tan similares para flujos y listas indica que nos falta alguna abstracción subyacente. Desafortunadamente, para explotar esta abstracción, necesitaremos ejercer un control más fino sobre el proceso de evaluación del que podemos en este momento. Discutiremos este punto más adelante al final de Section 3.5.4. En Section 4.2, desarrollaremos un marco que unifica listas y flujos.

56. Aunque `stream/car` y `stream/cdr` pueden definirse como procedimientos, `cons/stream` debe ser una forma especial. Si `cons/stream` fuera un procedimiento, entonces, según nuestro modelo de evaluación, evaluar `(cons/stream` ⟨*a*⟩ ⟨*b*⟩`)` causaría automáticamente que ⟨*b*$ \kern0.08em\rangle$ se evaluara, lo cual es precisamente lo que no queremos que suceda. Por la misma razón, `delay` debe ser una forma especial, aunque `force` puede ser un procedimiento ordinario.

57. Los números que se muestran aquí no aparecen realmente en la expresión retardada. Lo que realmente aparece es la expresión original, en un entorno en el que las variables están ligadas a los números apropiados. Por ejemplo, `(+ low 1)` con `low` ligado a 10,000 aparece realmente donde se muestra

10001.

58. Existen muchas implementaciones posibles de flujos además de la descrita en esta sección. La evaluación retardada, que es la clave para hacer que los flujos sean prácticos, era inherente al método de paso de parámetros *llamada por nombre* de Algol 60. El uso de este mecanismo para implementar flujos fue descrito por primera vez por Landin (1965). La evaluación retardada para flujos fue introducida en Lisp por Friedman y Wise (1976). En su implementación, `cons` siempre retrasa la evaluación de sus argumentos, de modo que las listas se comportan automáticamente como flujos. La optimización de memoización también se conoce como *llamada por necesidad*. La comunidad de Algol se referiría a nuestros objetos retardados originales como *thunks de llamada por nombre* y a las versiones optimizadas como *thunks de llamada por necesidad*.

59. Ejercicios como el Ejercicio 3.51 y el Ejercicio 3.52 son valiosos para probar nuestra comprensión de cómo funciona `delay`. Por otro lado, mezclar la evaluación retardada con la impresión—y, peor aún, con la asignación—es extremadamente confuso, y los instructores de cursos de lenguajes de programación han atormentado tradicionalmente a sus estudiantes con preguntas de examen como las de esta sección. No hace falta decir que escribir programas que dependen de tales sutilezas es un estilo de programación odioso. Parte del poder del procesamiento de flujos es que nos permite ignorar el orden en que los eventos ocurren realmente en nuestros programas. Desafortunadamente, esto es precisamente lo que no podemos permitirnos hacer en presencia de la asignación, que nos obliga a preocuparnos por el tiempo y el cambio.

60. Eratóstenes, un filósofo griego alejandrino del siglo III a. C., es famoso por dar la primera estimación precisa de la circunferencia de la Tierra, que calculó observando las sombras proyectadas al mediodía del día del solsticio de verano. El método de la criba de Eratóstenes, aunque antiguo, ha formado la base para el hardware de propósito especial de "cribas" que, hasta hace poco, eran las herramientas más poderosas existentes para localizar grandes números primos. Desde los años 70, sin embargo, estos métodos han sido superados por derivados de las técnicas probabilísticas discutidas en Section 1.2.6.

61. Hemos nombrado estas figuras en honor a Peter Henderson, quien fue la primera persona en mostrarnos diagramas de este tipo como una forma de pensar sobre el procesamiento de flujos. Cada línea sólida representa un flujo de valores que se transmite. La línea discontinua desde el `car` hasta el `cons` y el `filter` indica que este es un solo valor en lugar de un flujo.

62. Esto usa la versión generalizada de `stream/map` del Ejercicio 3.50.

63. Este último punto es muy sutil y se basa en el hecho de que $p_{n+1} \leq p_n^2$. (Aquí, p_k denota el k^{th} primo.) Estimaciones como estas son muy difíciles de establecer. La antigua prueba de Euclides de que hay un número infinito de primos muestra que $p_{n+1} \leq p_1 p_2 \cdots p_n + 1$, y no se probó ningún resultado sustancialmente mejor hasta 1851, cuando el matemático ruso P. L. Chebyshev estableció que $p_{n+1} \leq 2p_n$ para todo n. Este resultado, conjeturado originalmente en 1845, se conoce como la *hipótesis de Bertrand*. Se puede encontrar una prueba en la sección 22.3 de Hardy and Wright 1960.

64. Este ejercicio muestra cómo la llamada por necesidad está estrechamente relacionada con la memoización ordinaria como se describe en el Ejercicio 3.27. En ese ejercicio, usamos la asignación para construir explícitamente una tabla local. Nuestra optimización de flujo de llamada por necesidad construye efectivamente tal tabla automáticamente, almacenando valores en las partes previamente forzadas del flujo.

65. No podemos usar `let` para ligar la variable local `guesses`, porque el valor de `guesses` depende de `guesses` en sí. El Ejercicio 3.63 aborda por qué queremos una variable local aquí.

66. Como en Section 2.2.3, representamos un par de enteros como una lista en lugar de un par de Lisp.

67. Ver Ejercicio 3.68 para alguna idea de por qué elegimos esta descomposición.

68. La declaración precisa de la propiedad requerida en el orden de combinación es la siguiente: debe haber una función f de dos argumentos tal que el par correspondiente al elemento i del primer flujo y al elemento j del segundo flujo aparecerá como el elemento número $f(i,j)$ del flujo de salida. El truco de usar `interleave` para lograr esto nos lo mostró David Turner, quien lo empleó en el lenguaje KRC (Turner 1981).

69. Requeriremos que la función de ponderación sea tal que el peso de un par aumente a medida que nos movemos a lo largo de una fila o hacia abajo a lo largo de una columna de la matriz de pares.

70. Para citar el obituario de Ramanujan de G. H. Hardy (Hardy 1921): "Fue el Sr. Littlewood (creo) quien comentó que 'todo entero positivo era uno de sus amigos'. Recuerdo una vez que fui a verlo cuando estaba enfermo en Putney. Había viajado en el taxi número 1729, y comenté que el número me parecía bastante aburrido, y que esperaba que no fuera un presagio

desfavorable. 'No', respondió, 'es un número muy interesante; es el número más pequeño expresable como la suma de dos cubos de dos maneras diferentes'. " El truco de usar pares ponderados para generar los números de Ramanujan nos lo mostró Charles Leiserson.

71. Este procedimiento no se garantiza que funcione en todas las implementaciones de Scheme, aunque para cualquier implementación hay una variación simple que funcionará. El problema tiene que ver con diferencias sutiles en las formas en que las implementaciones de Scheme manejan las definiciones internas. (Ver Section 4.1.6.)

72. Esta es una pequeña reflexión, en Lisp, de las dificultades que los lenguajes fuertemente tipados convencionales como Pascal tienen para lidiar con los procedimientos de orden superior. En tales lenguajes, el programador debe especificar los tipos de datos de los argumentos y el resultado de cada procedimiento: número, valor lógico, secuencia, etc. En consecuencia, no podríamos expresar una abstracción como "mapear un procedimiento dado `proc` sobre todos los elementos de una secuencia" mediante un solo procedimiento de orden superior como `stream/map`. Más bien, necesitaríamos un procedimiento de mapeo diferente para cada combinación diferente de tipos de datos de argumento y resultado que pudiera especificarse para un `proc`. Mantener una noción práctica de "tipo de dato" en presencia de procedimientos de orden superior plantea muchos problemas difíciles. Una forma de lidiar con este problema se ilustra con el lenguaje ML (Gordon et al. 1979), cuyos "tipos de datos polimórficos" incluyen plantillas para transformaciones de orden superior entre tipos de datos. Además, los tipos de datos para la mayoría de los procedimientos en ML nunca son declarados explícitamente por el programador. En su lugar, ML incluye un mecanismo de *inferencia de tipos* que utiliza información del entorno para deducir los tipos de datos de los procedimientos recién definidos.

73. De manera similar en física, cuando observamos una partícula en movimiento, decimos que la posición (estado) de la partícula está cambiando. Sin embargo, desde la perspectiva de la línea de mundo de la partícula en el espacio-tiempo, no hay ningún cambio involucrado.

74. John Backus, el inventor de Fortran, le dio una gran visibilidad a la programación funcional cuando fue galardonado con el Premio Turing de la ACM en 1978. Su discurso de aceptación (Backus 1978) defendió enérgicamente el enfoque funcional. Se ofrece una buena visión general de la programación funcional en Henderson 1980 y en Darlington et al. 1982.

75. Observe que, para cualquier par de flujos, en general hay más de un orden de entrelazado aceptable. Por lo tanto, técnicamente, "fusionar" es una

relación en lugar de una función—la respuesta no es una función determinista de las entradas. Ya mencionamos (Nota 39) que el no determinismo es esencial cuando se trata de concurrencia. La relación de fusión ilustra el mismo no determinismo esencial, desde la perspectiva funcional. En Section 4.3, veremos el no determinismo desde otro punto de vista.

76. El modelo de objetos aproxima el mundo dividiéndolo en piezas separadas. El modelo funcional no modulariza a lo largo de los límites de los objetos. El modelo de objetos es útil cuando el estado no compartido de los "objetos" es mucho mayor que el estado que comparten. Un ejemplo de un lugar donde el punto de vista del objeto falla es la mecánica cuántica, donde pensar en las cosas como partículas individuales conduce a paradojas y confusiones. Unificar la visión del objeto con la visión funcional puede tener poco que ver con la programación, sino más bien con problemas epistemológicos fundamentales.

Abstracción Metalingüística

> … Es en las palabras donde está la magia—Abracadabra, Ábrete Sésamo, y el resto—pero las palabras mágicas en una historia no son mágicas en la siguiente. La verdadera magia es entender qué palabras funcionan, y cuándo, y para qué; el truco es aprender el truco.
>
> … Y esas palabras están hechas de las letras de nuestro alfabeto: un par de docenas de garabatos que podemos dibujar con la pluma. ¡Esta es la clave! ¡Y el tesoro, también, si tan solo pudiéramos ponerle las manos encima! ¡Es como si—como si la clave del tesoro *es* el tesoro!
>
> —John Barth, *Quimera*

En nuestro estudio del diseño de programas, hemos visto que los programadores expertos controlan la complejidad de sus diseños con las mismas técnicas generales utilizadas por los diseñadores de todos los sistemas complejos. Combinan elementos primitivos para formar objetos compuestos, abstraen objetos compuestos para formar bloques de construcción de nivel superior, y preservan la modularidad adoptando vistas a gran escala apropiadas de la estructura del sistema. Al ilustrar estas técnicas, hemos usado Lisp como un lenguaje para describir procesos y para construir objetos y procesos de datos computacionales para modelar fenómenos complejos en el mundo real. Sin embargo, a medida que nos enfrentamos a problemas cada vez más complejos, descubriremos que Lisp, o de hecho cualquier lenguaje de programación fijo, no es suficiente para nuestras necesidades. Debemos recurrir constantemente a nuevos lenguajes para expresar nuestras ideas más eficazmente. Establecer nuevos lenguajes es una estrategia poderosa para controlar la complejidad en el diseño de ingeniería; a menudo podemos mejorar nuestra capacidad para lidiar con un problema complejo adoptando un nuevo lenguaje que nos permita describir (y por lo tanto pensar sobre) el problema de una manera diferente, usando primitivas, medios de combinación y medios de abstracción que están particularmente bien adaptados al problema en cuestión.[1]

La programación está dotada de una multitud de lenguajes. Hay lenguajes físicos, como los lenguajes de máquina para computadoras particulares. Estos lenguajes se preocupan por la representación de datos y control en términos

de bits individuales de almacenamiento e instrucciones de máquina primitivas. El programador de lenguaje de máquina se preocupa por usar el hardware dado para erigir sistemas y utilidades para la implementación eficiente de cálculos con recursos limitados. Los lenguajes de alto nivel, erigidos sobre un sustrato de lenguaje de máquina, ocultan las preocupaciones sobre la representación de datos como colecciones de bits y la representación de programas como secuencias de instrucciones primitivas. Estos lenguajes tienen medios de combinación y abstracción, como la definición de procedimientos, que son apropiados para la organización a mayor escala de los sistemas.

La *abstracción metalingüística*—establecer nuevos lenguajes—juega un papel importante en todas las ramas del diseño de ingeniería. Es particularmente importante para la programación de computadoras, porque en la programación no solo podemos formular nuevos lenguajes sino que también podemos implementar estos lenguajes construyendo evaluadores. Un *evaluador* (o *intérprete*) para un lenguaje de programación es un procedimiento que, cuando se aplica a una expresión del lenguaje, realiza las acciones requeridas para evaluar esa expresión.

No es una exageración considerar esto como la idea más fundamental en programación:

> El evaluador, que determina el significado de las expresiones en un lenguaje de programación, es solo otro programa.

Apreciar este punto es cambiar nuestras imágenes de nosotros mismos como programadores. Llegamos a vernos como diseñadores de lenguajes, en lugar de solo usuarios de lenguajes diseñados por otros.

De hecho, podemos considerar casi cualquier programa como el evaluador de algún lenguaje. Por ejemplo, el sistema de manipulación de polinomios de Section 2.5.3 encarna las reglas de la aritmética de polinomios y las implementa en términos de operaciones sobre datos estructurados en listas. Si aumentamos este sistema con procedimientos para leer e imprimir expresiones polinómicas, tenemos el núcleo de un lenguaje de propósito especial para tratar problemas de matemáticas simbólicas. El simulador de lógica digital de Section 3.3.4 y el propagador de restricciones de Section 3.3.5 son lenguajes legítimos por derecho propio, cada uno con sus propias primitivas, medios de combinación y medios de abstracción. Visto desde esta perspectiva, la tecnología para hacer frente a los sistemas informáticos a gran escala se fusiona con la tecnología para construir nuevos lenguajes informáticos, y la informática misma se convierte en nada más (y nada menos) que la disciplina de construir lenguajes descriptivos apropiados.

Ahora nos embarcamos en un recorrido por la tecnología mediante la cual se establecen los lenguajes en términos de otros lenguajes. En este capítulo usaremos Lisp como base, implementando evaluadores como procedimientos Lisp. Lisp es particularmente adecuado para esta tarea, debido a su capacidad para representar y manipular expresiones simbólicas. Daremos el primer paso para comprender cómo se implementan los lenguajes construyendo un evaluador para Lisp mismo. El lenguaje implementado por nuestro evaluador será un subconjunto del dialecto Scheme de Lisp que usamos en este libro. Aunque el evaluador descrito en este capítulo está escrito para un dialecto particular de Lisp, contiene la estructura esencial de un evaluador para cualquier lenguaje orientado a expresiones diseñado para escribir programas para una máquina secuencial. (De hecho, la mayoría de los procesadores de lenguaje contienen, en lo profundo de ellos, un pequeño evaluador "Lisp".) El evaluador ha sido simplificado para fines de ilustración y discusión, y se han omitido algunas características que sería importante incluir en un sistema Lisp de calidad de producción. Sin embargo, este evaluador simple es adecuado para ejecutar la mayoría de los programas de este libro.[2]

Una ventaja importante de hacer que el evaluador sea accesible como un programa Lisp es que podemos implementar reglas de evaluación alternativas describiéndolas como modificaciones al programa del evaluador. Un lugar donde podemos usar este poder con buenos resultados es para obtener un control extra sobre las formas en que los modelos computacionales encarnan la noción de tiempo, que fue tan central en la discusión del Capítulo 3. Allí, mitigamos algunas de las complejidades del estado y la asignación usando flujos para desacoplar la representación del tiempo en el mundo del tiempo en la computadora. Sin embargo, nuestros programas de flujos a veces eran engorrosos, porque estaban restringidos por la evaluación de orden aplicativo de Scheme. En Section 4.2, cambiaremos el lenguaje subyacente para proporcionar un enfoque más elegante, modificando el evaluador para proporcionar una *evaluación de orden normal*.

El Section 4.3 implementa un cambio lingüístico más ambicioso, mediante el cual las expresiones tienen muchos valores, en lugar de un solo valor. En este lenguaje de *computación no determinista*, es natural expresar procesos que generan todos los valores posibles para las expresiones y luego buscan aquellos valores que satisfacen ciertas restricciones. En términos de modelos de computación y tiempo, esto es como tener el tiempo ramificándose en un conjunto de "futuros posibles" y luego buscar líneas de tiempo apropiadas. Con nuestro evaluador no determinista, el seguimiento de múltiples valores y la realización de búsquedas son manejados automáticamente por el mecanismo subyacente del lenguaje.

En Section 4.4 implementamos un lenguaje de *programación lógica* en el que

el conocimiento se expresa en términos de relaciones, en lugar de en términos de cálculos con entradas y salidas. Aunque esto hace que el lenguaje sea drásticamente diferente de Lisp, o de hecho de cualquier lenguaje convencional, veremos que el evaluador de programación lógica comparte la estructura esencial del evaluador de Lisp.

El Evaluador Metacircular

Nuestro evaluador para Lisp se implementará como un programa Lisp. Puede parecer circular pensar en evaluar programas Lisp usando un evaluador que está implementado en Lisp. Sin embargo, la evaluación es un proceso, por lo que es apropiado describir el proceso de evaluación usando Lisp, que, después de todo, es nuestra herramienta para describir procesos.[3] Se dice que un evaluador que está escrito en el mismo lenguaje que evalúa es *metacircular*.

El evaluador metacircular es esencialmente una formulación en Scheme del modelo de evaluación de entorno descrito en Section 3.2. Recordemos que el modelo tiene dos partes básicas:

1. Para evaluar una combinación (una expresión compuesta que no sea una forma especial), evaluar las subexpresiones y luego aplicar el valor de la subexpresión del operador a los valores de las subexpresiones del operando.

2. Para aplicar un procedimiento compuesto a un conjunto de argumentos, evalúe el cuerpo del procedimiento en un nuevo entorno. Para construir este entorno, extienda el entorno parte del objeto de procedimiento por un marco en el que los parámetros formales del procedimiento están ligados a los argumentos a los que se aplica el procedimiento.

Estas dos reglas describen la esencia del proceso de evaluación, un ciclo básico en el que las expresiones a evaluar en entornos se reducen a procedimientos que se aplicarán a argumentos, que a su vez se reducen a nuevas expresiones que se evaluarán en nuevos entornos, y así sucesivamente, hasta que llegamos a los símbolos, cuyos valores se buscan en el entorno, y a los procedimientos primitivos, que se aplican directamente (ver Figura 4.1).[4] Este ciclo de evaluación se plasmará en la interacción entre los dos procedimientos críticos del evaluador, `eval` y `apply`, que se describen en Section 4.1.1 (ver Figura 4.1).

La implementación del evaluador dependerá de procedimientos que definan la *sintaxis* de las expresiones a evaluar. Usaremos la abstracción de datos para

hacer que el evaluador sea independiente de la representación del lenguaje. Por ejemplo, en lugar de comprometernos con la elección de que una asignación se represente mediante una lista que comienza con el símbolo `set!`, usamos un predicado abstracto `assignment?` para probar una asignación, y usamos selectores abstractos `assignment/variable` y `assignment/value` para acceder a las partes de una asignación. La implementación de las expresiones se describirá en detalle en Section 4.1.2. También hay operaciones, descritas en Section 4.1.3, que especifican la representación de procedimientos y entornos. Por ejemplo, `make/procedure` construye procedimientos compuestos, `lookup/variable/value` accede a los valores de las variables y `apply/primitive/procedure` aplica un procedimiento primitivo a una lista dada de argumentos.

Figura 4.1: El ciclo `eval-apply` expone la esencia de un lenguaje de computadora.

El Núcleo del Evaluador

El proceso de evaluación puede describirse como la interacción entre dos procedimientos: `eval` y `apply`.

Eval

`eval` toma como argumentos una expresión y un entorno. Clasifica la expresión y dirige su evaluación. `eval` está estructurado como un análisis de casos de la sintaxis de la expresión a evaluar. Para mantener el procedimiento general, expresamos la determinación del tipo de una expresión de forma abstracta, sin comprometernos con ninguna representación particular para los diversos tipos de expresiones. Cada tipo de expresión tiene un predicado que lo prueba y un medio abstracto para seleccionar sus partes. Esta *sintaxis abstracta* facilita ver cómo podemos cambiar la sintaxis del lenguaje usando el mismo evaluador, pero con una colección diferente de procedimientos de sintaxis.

Expresiones primitivas

- Para las expresiones que se autoevalúan, como los números, `eval` devuelve la expresión misma.
- `eval` debe buscar las variables en el entorno para encontrar sus valores.

Formas especiales

- Para las expresiones citadas, `eval` devuelve la expresión que fue citada.

- Una asignación a (o una definición de) una variable debe llamar recursivamente a `eval` para calcular el nuevo valor que se asociará con la variable. El entorno debe modificarse para cambiar (o crear) la ligadura de la variable.

- Una expresión `if` requiere un procesamiento especial de sus partes, para evaluar el consecuente si el predicado es verdadero, y en caso contrario evaluar la alternativa.

- Una expresión `lambda` debe transformarse en un procedimiento aplicable empaquetando los parámetros y el cuerpo especificados por la expresión `lambda` con el entorno de la evaluación.

- Una expresión `begin` requiere evaluar su secuencia de expresiones en el orden en que aparecen.

- Un análisis de casos (`cond`) se transforma en un nido de expresiones `if` y luego se evalúa.

Combinaciones

- Para una aplicación de procedimiento, `eval` debe evaluar recursivamente la parte del operador y los operandos de la combinación. El procedimiento y los argumentos resultantes se pasan a `apply`, que maneja la aplicación del procedimiento real.

Aquí está la definición de `eval`:

(define (eval exp env) (cond ((self-evaluating? exp) exp) ((variable? exp) (lookup-variable-value exp env)) ((quoted? exp) (text-of-quotation exp)) ((assignment? exp) (eval-assignment exp env)) ((definition? exp) (eval-definition exp env)) ((if? exp) (eval-if exp env)) ((lambda? exp) (make-procedure (lambda-parameters exp) (lambda-body exp) env)) ((begin? exp) (eval-sequence (begin-actions exp) env)) ((cond? exp) (eval (cond->if exp) env)) ((application? exp) (apply (eval (operator exp) env) (list-of-values (operands exp) env))) (else (error "Unknown expression type: EVAL" exp))))

Para mayor claridad, `eval` se ha implementado como un análisis de casos usando `cond`. La desventaja de esto es que nuestro procedimiento solo maneja unos pocos tipos distinguibles de expresiones, y no se pueden definir nuevos sin editar la definición de `eval`. En la mayoría de las implementaciones de Lisp, el despacho sobre el tipo de una expresión se realiza en un estilo dirigido por datos. Esto permite al usuario agregar nuevos

tipos de expresiones que `eval` puede distinguir, sin modificar la definición de `eval` en sí. (Ver Ejercicio 4.3.)

Apply

`apply` toma dos argumentos, un procedimiento y una lista de argumentos a los que se debe aplicar el procedimiento. `apply` clasifica los procedimientos en dos tipos: llama a `apply/primitive/procedure` para aplicar primitivas; aplica procedimientos compuestos evaluando secuencialmente las expresiones que componen el cuerpo del procedimiento. El entorno para la evaluación del cuerpo de un procedimiento compuesto se construye extendiendo el entorno base llevado por el procedimiento para incluir un marco que liga los parámetros del procedimiento a los argumentos a los que se va a aplicar el procedimiento. Aquí está la definición de `apply`:

(define (apply procedure arguments) (cond ((primitive-procedure? procedure) (apply-primitive-procedure procedure arguments)) ((compound-procedure? procedure) (eval-sequence (procedure-body procedure) (extend-environment (procedure-parameters procedure) arguments (procedure-environment procedure)))) (else (error "Unknown procedure type: APPLY" procedure))))

Argumentos de procedimiento

Cuando `eval` procesa una aplicación de procedimiento, utiliza `list/of/values` para producir la lista de argumentos a los que se aplicará el procedimiento. `list/of/values` toma como argumento los operandos de la combinación. Evalúa cada operando y devuelve una lista de los valores correspondientes:[5]

(define (list-of-values exps env) (if (no-operands? exps) '() (cons (eval (first-operand exps) env) (list-of-values (rest-operands exps) env))))

Condicionales

`eval/if` evalúa la parte del predicado de una expresión `if` en el entorno dado. Si el resultado es verdadero, `eval/if` evalúa el consecuente, de lo contrario evalúa la alternativa:

(define (eval-if exp env) (if (true? (eval (if-predicate exp) env)) (eval (if-consequent exp) env) (eval (if-alternative exp) env)))

El uso de `true?` en `eval/if` resalta el problema de la conexión entre un

lenguaje implementado y un lenguaje de implementación. El `if/predicate` se evalúa en el lenguaje que se está implementando y, por lo tanto, produce un valor en ese lenguaje. El predicado del intérprete `true?` traduce ese valor a un valor que puede ser probado por el `if` en el lenguaje de implementación: la representación metacircular de la verdad podría no ser la misma que la del Scheme subyacente.[6]

Secuencias

`eval/sequence` es utilizado por `apply` para evaluar la secuencia de expresiones en el cuerpo de un procedimiento y por `eval` para evaluar la secuencia de expresiones en una expresión `begin`. Toma como argumentos una secuencia de expresiones y un entorno, y evalúa las expresiones en el orden en que aparecen. El valor devuelto es el valor de la expresión final.

(define (eval-sequence exps env) (cond ((last-exp? exps) (eval (first-exp exps) env)) (else (eval (first-exp exps) env) (eval-sequence (rest-exps exps) env))))

Asignaciones y definiciones

El siguiente procedimiento maneja las asignaciones a las variables. Llama a `eval` para encontrar el valor a asignar y transmite la variable y el valor resultante a `set/variable/value!` para que se instale en el entorno designado.

(define (eval-assignment exp env) (set-variable-value! (assignment-variable exp) (eval (assignment-value exp) env) env) 'ok)

Las definiciones de variables se manejan de manera similar.[7]

(define (eval-definition exp env) (define-variable! (definition-variable exp) (eval (definition-value exp) env) env) 'ok)

Hemos elegido aquí devolver el símbolo `ok` como el valor de una asignación o una definición.[8]

> **Ejercicio 4.1:** Nótese que no podemos decir si el evaluador metacircular evalúa los operandos de izquierda a derecha o de derecha a izquierda. Su orden de evaluación se hereda del Lisp subyacente: Si los argumentos de `cons` en `list/of/values` se evalúan de izquierda a derecha, entonces `list/of/values` evaluará los operandos de izquierda a derecha; y si los argumentos de `cons` se evalúan de derecha a izquierda, entonces `list/of/values` evaluará los operandos de derecha a izquierda.

Escriba una versión de `list/of/values` que evalúe los operandos de izquierda a derecha independientemente del orden de evaluación en el Lisp subyacente. Escriba también una versión de `list/of/values` que evalúe los operandos de derecha a izquierda.

Representando Expresiones

El evaluador recuerda al programa de diferenciación simbólica discutido en Section 2.3.2. Ambos programas operan sobre expresiones simbólicas. En ambos programas, el resultado de operar sobre una expresión compuesta se determina operando recursivamente sobre las partes de la expresión y combinando los resultados de una manera que depende del tipo de la expresión. En ambos programas usamos la abstracción de datos para desacoplar las reglas generales de operación de los detalles de cómo se representan las expresiones. En el programa de diferenciación esto significaba que el mismo procedimiento de diferenciación podía tratar con expresiones algebraicas en forma de prefijo, en forma de infijo o en alguna otra forma. Para el evaluador, esto significa que la sintaxis del lenguaje que se está evaluando está determinada únicamente por los procedimientos que clasifican y extraen partes de las expresiones.

Aquí está la especificación de la sintaxis de nuestro lenguaje:

- Los únicos elementos que se autoevalúan son los números y las cadenas:

  (define (self-evaluating? exp) (cond ((number? exp) true) ((string? exp) true) (else false)))

- Las variables se representan mediante símbolos:

  (define (variable? exp) (symbol? exp))

- Las citas tienen la forma `(quote` ⟨*texto/de/la/cita*⟩`)`:[9]

  (define (quoted? exp) (tagged-list? exp 'quote)) (define (text-of-quotation exp) (cadr exp))

  `quoted?` se define en términos del procedimiento `tagged/list?`, que identifica las listas que comienzan con un símbolo designado:

  (define (tagged-list? exp tag) (if (pair? exp) (eq? (car exp) tag) false))

- Las asignaciones tienen la forma `(set!` ⟨*var*⟩ ⟨*valor*⟩`)`:

  (define (assignment? exp) (tagged-list? exp 'set!)) (define (assignment-

variable exp) (cadr exp)) (define (assignment-value exp) (caddr exp))

- Las definiciones tienen la forma

  (define ⟨ *var* ⟩ ⟨ *valor* ⟩)

  o la forma

  (define (⟨ *var* ⟩ ⟨ *parámetro* $\color{SchemeDark}_{\hbox{\ttfamily \scriptsize 1}}\rangle$ … ⟨ *parámetro* $\color{SchemeDark} _{\hbox{\ttfamily\itshape\scriptsize n}}\rangle$) ⟨ *cuerpo* ⟩)

  La última forma (definición de procedimiento estándar) es azúcar sintáctico para

  (define ⟨ *var* ⟩ (lambda (⟨ *parámetro* $\color{SchemeDark} _{\hbox{\ttfamily\scriptsize 1}}\rangle$ … ⟨ *parámetro* $ \color{SchemeDark}_{\hbox{\ttfamily\itshape\scriptsize n}}\rangle$) ⟨ *cuerpo* ⟩))

  Los procedimientos de sintaxis correspondientes son los siguientes:

  (define (definition? exp) (tagged-list? exp 'define)) (define (definition-variable exp) (if (symbol? (cadr exp)) (cadr exp) (caadr exp))) (define (definition-value exp) (if (symbol? (cadr exp)) (caddr exp) (make-lambda (cdadr exp) ; parámetros formales (cddr exp)))) ; cuerpo

- Las expresiones `lambda` son listas que comienzan con el símbolo `lambda`:

  (define (lambda? exp) (tagged-list? exp 'lambda)) (define (lambda-parameters exp) (cadr exp)) (define (lambda-body exp) (cddr exp))

  También proporcionamos un constructor para las expresiones `lambda`, que es utilizado por `definition/value`, arriba:

  (define (make-lambda parameters body) (cons 'lambda (cons parameters body)))

- Los condicionales comienzan con `if` y tienen un predicado, un consecuente y una alternativa (opcional). Si la expresión no tiene parte alternativa, proporcionamos `false` como alternativa.[10]

  (define (if? exp) (tagged-list? exp 'if)) (define (if-predicate exp) (cadr exp)) (define (if-consequent exp) (caddr exp)) (define (if-alternative exp) (if (not (null? (cdddr exp))) (cadddr exp) 'false))

También proporcionamos un constructor para las expresiones `if`, que será utilizado por `cond/>if` para transformar las expresiones `cond` en expresiones `if`:

(define (make-if predicate consequent alternative) (list 'if predicate consequent alternative))

- `begin` empaqueta una secuencia de expresiones en una sola expresión. Incluimos operaciones de sintaxis en las expresiones `begin` para extraer la secuencia real de la expresión `begin`, así como selectores que devuelven la primera expresión y el resto de las expresiones de la secuencia.[11]

  (define (begin? exp) (tagged-list? exp 'begin)) (define (begin-actions exp) (cdr exp)) (define (last-exp? seq) (null? (cdr seq))) (define (first-exp seq) (car seq)) (define (rest-exps seq) (cdr seq))

  También incluimos un constructor `sequence/>exp` (para ser usado por `cond/>if`) que transforma una secuencia en una sola expresión, usando `begin` si es necesario:

  (define (sequence->exp seq) (cond ((null? seq) seq) ((last-exp? seq) (first-exp seq)) (else (make-begin seq)))) (define (make-begin seq) (cons 'begin seq))

- Una aplicación de procedimiento es cualquier expresión compuesta que no sea uno de los tipos de expresión anteriores. El `car` de la expresión es el operador, y el `cdr` es la lista de operandos:

  (define (application? exp) (pair? exp)) (define (operator exp) (car exp)) (define (operands exp) (cdr exp)) (define (no-operands? ops) (null? ops)) (define (first-operand ops) (car ops)) (define (rest-operands ops) (cdr ops))

Expresiones derivadas

Algunas formas especiales en nuestro lenguaje pueden definirse en términos de expresiones que involucran otras formas especiales, en lugar de ser implementadas directamente. Un ejemplo es `cond`, que puede implementarse como un nido de expresiones `if`. Por ejemplo, podemos reducir el problema de evaluar la expresión

(cond ((> x 0) x) ((= x 0) (display 'zero) 0) (else (- x)))

al problema de evaluar la siguiente expresión que involucra expresiones `if` y

`begin`:

(if (> x 0) x (if (= x 0) (begin (display 'zero) 0) (- x)))

Implementar la evaluación de `cond` de esta manera simplifica el evaluador porque reduce el número de formas especiales para las cuales el proceso de evaluación debe especificarse explícitamente.

Incluimos procedimientos de sintaxis que extraen las partes de una expresión `cond`, y un procedimiento `cond/>if` que transforma las expresiones `cond` en expresiones `if`. Un análisis de casos comienza con `cond` y tiene una lista de cláusulas predicado-acción. Una cláusula es una cláusula `else` si su predicado es el símbolo `else`.[12]

(define (cond? exp) (tagged-list? exp 'cond)) (define (cond-clauses exp) (cdr exp)) (define (cond-else-clause? clause) (eq? (cond-predicate clause) 'else)) (define (cond-predicate clause) (car clause)) (define (cond-actions clause) (cdr clause)) (define (cond->if exp) (expand-clauses (cond-clauses exp))) (define (expand-clauses clauses) (if (null? clauses) 'false ; no `else` clause (let ((first (car clauses)) (rest (cdr clauses))) (if (cond-else-clause? first) (if (null? rest) (sequence->exp (cond-actions first)) (error "ELSE clause isn't last: COND->IF" clauses)) (make-if (cond-predicate first) (sequence->exp (cond-actions first)) (expand-clauses rest))))))

Las expresiones (como `cond`) que elegimos implementar como transformaciones sintácticas se llaman *expresiones derivadas*. Las expresiones `let` también son expresiones derivadas (ver Ejercicio 4.6).[13]

> **Ejercicio 4.2:** Louis Reasoner planea reordenar las cláusulas `cond` en `eval` para que la cláusula para las aplicaciones de procedimientos aparezca antes que la cláusula para las asignaciones. Argumenta que esto hará que el intérprete sea más eficiente: dado que los programas suelen contener más aplicaciones que asignaciones, definiciones, etc., su `eval` modificado generalmente comprobará menos cláusulas que el `eval` original antes de identificar el tipo de una expresión.
>
> 1. ¿Qué hay de malo en el plan de Louis? (Pista: ¿Qué hará el evaluador de Louis con la expresión `(define x 3)`?)
>
> 2. Louis está molesto porque su plan no funcionó. Está dispuesto a hacer lo que sea para que su evaluador reconozca las aplicaciones de procedimientos antes de que compruebe la mayoría de los otros tipos de expresiones. Ayúdale cambiando la sintaxis del lenguaje evaluado para que las aplicaciones de procedimientos comiencen con `call`. Por ejemplo, en lugar de `(factorial 3)` ahora tendremos que escribir

`(call factorial 3)` y en lugar de `(+ 1 2)` tendremos que escribir `(call + 1 2)`.

Ejercicio 4.3: Vuelva a escribir `eval` para que el despacho se haga en estilo dirigido por datos. Compare esto con el procedimiento de diferenciación dirigido por datos del Ejercicio 2.73. (Puede usar el `car` de una expresión compuesta como el tipo de la expresión, como es apropiado para la sintaxis implementada en esta sección.)

Ejercicio 4.4: Recuerde las definiciones de las formas especiales `and` y `or` del Capítulo 1:

- `and`: Las expresiones se evalúan de izquierda a derecha. Si alguna expresión se evalúa como falsa, se devuelve falso; las expresiones restantes no se evalúan. Si todas las expresiones se evalúan como valores verdaderos, el valor de la última expresión se devuelve. Si no hay expresiones, se devuelve verdadero.

- `or`: Las expresiones se evalúan de izquierda a derecha. Si alguna expresión se evalúa como un valor verdadero, se devuelve ese valor; las expresiones restantes no se evalúan. Si todas las expresiones se evalúan como falsas, o si no hay expresiones, entonces se devuelve falso.

Instale `and` y `or` como nuevas formas especiales para el evaluador definiendo procedimientos de sintaxis y procedimientos de evaluación apropiados `eval/and` y `eval/or`. Alternativamente, muestre cómo implementar `and` y `or` como expresiones derivadas.

Ejercicio 4.5: Scheme permite una sintaxis adicional para las cláusulas `cond`, `(`⟨*test*⟩ `=>` ⟨*recipient*⟩`)`. Si ⟨*test*$\kern0.08em\rangle$ se evalúa como un valor verdadero, entonces se evalúa ⟨*recipient*$\kern0.08em \rangle$. Su valor debe ser un procedimiento de un argumento; este procedimiento se invoca entonces sobre el valor de ⟨*test*$\kern0.08em \rangle$, y el resultado se devuelve como el valor de la expresión `cond`. Por ejemplo

(cond ((assoc ’b ’((a 1) (b 2))) = > cadr) (else false))

devuelve 2. Modifique el manejo de `cond` para que admita esta sintaxis extendida.

Ejercicio 4.6: Las expresiones `let` son expresiones derivadas, porque

(let ((< *var* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize

1}}\rangle$ ⟨ *exp* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$) … (⟨ *var* $\color{SchemeDark}_{\hbox{\ttfamily \itshape\scriptsize n}}\rangle$ ⟨ *exp* $\color{SchemeDark} _{\hbox{\ttfamily\itshape\scriptsize n}}\rangle$)) ⟨ *cuerpo* ⟩)

es equivalente a

((lambda (⟨ *var* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$ … ⟨ *var* $\color{SchemeDark}_{\hbox{\ttfamily\itshape \scriptsize n}}\rangle$) ⟨ *cuerpo* ⟩) ⟨ *exp* $\color{SchemeDark} _{\hbox{\ttfamily\scriptsize 1}}\rangle$ … ⟨ *exp* $ \color{SchemeDark}_{\hbox{\ttfamily\itshape\scriptsize n}}\rangle$)

Implemente una transformación sintáctica `let/>combination` que reduzca la evaluación de expresiones `let` a la evaluación de combinaciones del tipo mostrado anteriormente, y agregue la cláusula apropiada a `eval` para manejar las expresiones `let`.

Ejercicio 4.7: `let*` es similar a `let`, excepto que las ligaduras de las variables `let*` se realizan secuencialmente de izquierda a derecha, y cada ligadura se hace en un entorno en el que todas las ligaduras anteriores son visibles. Por ejemplo

(let* ((x 3) (y (+ x 2)) (z (+ x y 5))) (* x z))

devuelve 39. Explique cómo una expresión `let*` puede reescribirse como un conjunto de expresiones `let` anidadas, y escriba un procedimiento `let*/>nested/lets` que realice esta transformación. Si ya hemos implementado `let` (Ejercicio 4.6) y queremos extender el evaluador para manejar `let*`, ¿es suficiente agregar una cláusula a `eval` cuya acción sea

(eval (let*->nested-lets exp) env)

o debemos expandir explícitamente `let*` en términos de expresiones no derivadas?

Ejercicio 4.8: El "`let` con nombre" es una variante de `let` que tiene la forma

(let ⟨ *var* ⟩ ⟨ *bindings* ⟩ ⟨ *body* ⟩)

Los ⟨*bindings*$\kern0.08em\rangle$ y ⟨*body*$\kern0.08em\rangle$ son como en un `let` ordinario, excepto que ⟨*var*$\kern0.08em\rangle$ está ligado dentro de ⟨*body*$\kern0.08em\rangle$ a un procedimiento cuyo cuerpo es ⟨*body*$\kern0.08em\rangle$ y cuyos parámetros son las

variables en los ⟨*bindings*$\kern0.08em\rangle$. Por lo tanto, uno puede ejecutar repetidamente el ⟨*body*$\kern0.08em\rangle$ invocando el procedimiento llamado ⟨*var*$\kern0.08em\rangle$. Por ejemplo, el procedimiento iterativo de Fibonacci (Section 1.2.2) puede reescribirse usando un `let` con nombre de la siguiente manera:

(define (fib n) (let fib-iter ((a 1) (b 0) (count n)) (if (= count 0) b (fib-iter (+ a b) a (- count 1)))))

Modifique `let/>combination` del Ejercicio 4.6 para que también admita el `let` con nombre.

Ejercicio 4.9: Muchos lenguajes admiten una variedad de construcciones de iteración, como `do`, `for`, `while` y `until`. En Scheme, los procesos iterativos pueden expresarse en términos de llamadas a procedimientos ordinarios, por lo que las construcciones de iteración especiales no proporcionan ninguna ganancia esencial en el poder computacional. Por otro lado, tales construcciones son a menudo convenientes. Diseñe algunas construcciones de iteración, dé ejemplos de su uso y muestre cómo implementarlas como expresiones derivadas.

Ejercicio 4.10: Mediante el uso de la abstracción de datos, pudimos escribir un procedimiento `eval` que es independiente de la sintaxis particular del lenguaje a evaluar. Para ilustrar esto, diseñe e implemente una nueva sintaxis para Scheme modificando los procedimientos de esta sección, sin cambiar `eval` o `apply`.

Estructuras de Datos del Evaluador

Además de definir la sintaxis externa de las expresiones, la implementación del evaluador también debe definir las estructuras de datos que el evaluador manipula internamente, como parte de la ejecución de un programa, como la representación de procedimientos y entornos y la representación de verdadero y falso.

Prueba de predicados

Para los condicionales, aceptamos como verdadero todo lo que no sea el objeto explícito `false`.

(define (true? x) (not (eq? x false))) (define (false? x) (eq? x false))

Representando procedimientos

Para manejar primitivas, asumimos que tenemos disponibles los siguientes procedimientos:

- `(apply/primitive/procedure` ⟨*proc*⟩ ⟨*args*⟩`)`

  aplica el procedimiento primitivo dado a los valores de los argumentos en la lista ⟨*args*$\kern0.08em\rangle$ y devuelve el resultado de la aplicación.

- `(primitive/procedure?` ⟨*proc*⟩`)`

  prueba si ⟨*proc*$\kern0.08em\rangle$ es un procedimiento primitivo.

Estos mecanismos para manejar primitivas se describen con más detalle en la Sección 4.1.4.

Los procedimientos compuestos se construyen a partir de parámetros, cuerpos de procedimiento y entornos usando el constructor `make/procedure`:

(define (make-procedure parameters body env) (list 'procedure parameters body env)) (define (compound-procedure? p) (tagged-list? p 'procedure)) (define (procedure-parameters p) (cadr p)) (define (procedure-body p) (caddr p)) (define (procedure-environment p) (cadddr p))

Operaciones sobre Entornos

El evaluador necesita operaciones para manipular entornos. Como se explicó en Section 3.2, un entorno es una secuencia de marcos, donde cada marco es una tabla de ligaduras que asocian variables con sus valores correspondientes. Usamos las siguientes operaciones para manipular entornos:

- `(lookup/variable/value` ⟨*var*⟩ ⟨*env*⟩`)`

  devuelve el valor que está ligado al símbolo ⟨*var*$\kern0.08em\rangle$ en el entorno ⟨*env*$\kern0.08em\rangle$, o señala un error si la variable no está ligada.

- `(extend/environment` ⟨*variables*⟩ ⟨*values*⟩ ⟨*base/env*⟩`)`

  devuelve un nuevo entorno, que consiste en un nuevo marco en el que los símbolos de la lista ⟨*variables*$\kern0.08em\rangle$ están ligados a los elementos correspondientes de la lista ⟨*values*$\kern0.08em\rangle$, donde el entorno envolvente es el entorno ⟨*base-env*$\kern0.08em\rangle$.

- `(define/variable! ⟨var⟩ ⟨value⟩ ⟨env⟩)`

  añade al primer marco del entorno ⟨*env*$\kern0.08em\rangle$ una nueva ligadura que asocia la variable ⟨*var*$\kern0.08em\rangle$ con el valor ⟨*value*$\kern0.08em\rangle$.

- `(set/variable/value! ⟨var⟩ ⟨value⟩ ⟨env⟩)`

  cambia la ligadura de la variable ⟨*var*$\kern0.08em\rangle$ en el entorno ⟨*env*$\kern0.08em\rangle$ de modo que la variable ahora esté ligada al valor ⟨*value*$\kern0.08em\rangle$, o señala un error si la variable no está ligada.

Para implementar estas operaciones, representamos un entorno como una lista de marcos. El entorno envolvente de un entorno es el `cdr` de la lista. El entorno vacío es simplemente la lista vacía.

(define (enclosing-environment env) (cdr env)) (define (first-frame env) (car env)) (define the-empty-environment '())

Cada marco de un entorno se representa como un par de listas: una lista de las variables ligadas en ese marco y una lista de los valores asociados.[14]

(define (make-frame variables values) (cons variables values)) (define (frame-variables frame) (car frame)) (define (frame-values frame) (cdr frame)) (define (add-binding-to-frame! var val frame) (set-car! frame (cons var (car frame))) (set-cdr! frame (cons val (cdr frame))))

Para extender un entorno con un nuevo marco que asocie variables con valores, hacemos un marco que consiste en la lista de variables y la lista de valores, y lo adjuntamos al entorno. Señalamos un error si el número de variables no coincide con el número de valores.

(define (extend-environment vars vals base-env) (if (= (length vars) (length vals)) (cons (make-frame vars vals) base-env) (if (< (length vars) (length vals)) (error "Too many arguments supplied" vars vals) (error "Too few arguments supplied" vars vals))))

Para buscar una variable en un entorno, recorremos la lista de variables en el primer marco. Si encontramos la variable deseada, devolvemos el elemento correspondiente en la lista de valores. Si no encontramos la variable en el marco actual, buscamos en el entorno envolvente, y así sucesivamente. Si llegamos al entorno vacío, señalamos un error de "variable no ligada".

(define (lookup-variable-value var env) (define (env-loop env) (define (scan

vars vals) (cond ((null? vars) (env-loop (enclosing-environment env))) ((eq? var (car vars)) (car vals)) (else (scan (cdr vars) (cdr vals))))) (if (eq? env the-empty-environment) (error "Unbound variable" var) (let ((frame (first-frame env))) (scan (frame-variables frame) (frame-values frame))))) (env-loop env))

Para establecer una variable a un nuevo valor en un entorno especificado, buscamos la variable, al igual que en `lookup/variable/value`, y cambiamos el valor correspondiente cuando la encontramos.

(define (set-variable-value! var val env) (define (env-loop env) (define (scan vars vals) (cond ((null? vars) (env-loop (enclosing-environment env))) ((eq? var (car vars)) (set-car! vals val)) (else (scan (cdr vars) (cdr vals))))) (if (eq? env the-empty-environment) (error "Unbound variable: SET!" var) (let ((frame (first-frame env))) (scan (frame-variables frame) (frame-values frame))))) (env-loop env))

Para definir una variable, buscamos en el primer marco una ligadura para la variable, y cambiamos la ligadura si existe (igual que en `set/variable/value!`). Si no existe tal ligadura, adjuntamos una al primer marco.

(define (define-variable! var val env) (let ((frame (first-frame env))) (define (scan vars vals) (cond ((null? vars) (add-binding-to-frame! var val frame)) ((eq? var (car vars)) (set-car! vals val)) (else (scan (cdr vars) (cdr vals))))) (scan (frame-variables frame) (frame-values frame))))

El método descrito aquí es solo una de las muchas formas plausibles de representar entornos. Dado que usamos la abstracción de datos para aislar el resto del evaluador de la elección detallada de la representación, podríamos cambiar la representación del entorno si quisiéramos. (Ver Ejercicio 4.11.) En un sistema Lisp de calidad de producción, la velocidad de las operaciones de entorno del evaluador, especialmente la de la búsqueda de variables, tiene un gran impacto en el rendimiento del sistema. La representación descrita aquí, aunque conceptualmente simple, no es eficiente y normalmente no se usaría en un sistema de producción.[15]

> **Ejercicio 4.11:** En lugar de representar un marco como un par de listas, podemos representar un marco como una lista de ligaduras, donde cada ligadura es un par nombre-valor. Vuelva a escribir las operaciones de entorno para usar esta representación alternativa.
>
> **Ejercicio 4.12:** Los procedimientos `set/variable/value!`, `define/variable!` y `lookup/variable/value` pueden expresarse en términos de procedimientos más abstractos para recorrer la estructura del entorno. Defina abstracciones que capture los patrones comunes y redefina los tres procedimientos en términos de estas abstracciones.

Ejercicio 4.13: Scheme nos permite crear nuevas ligaduras para las variables por medio de `define`, pero no proporciona ninguna forma de deshacerse de las ligaduras. Implemente para el evaluador una forma especial `make/unbound!` que elimine la ligadura de un símbolo dado del entorno en el que se evalúa la expresión `make/unbound!`. Este problema no está completamente especificado. Por ejemplo, ¿deberíamos eliminar solo la ligadura en el primer marco del entorno? Complete la especificación y justifique cualquier elección que haga.

Ejecutando el Evaluador como un Programa

Dado el evaluador, tenemos en nuestras manos una descripción (expresada en Lisp) del proceso mediante el cual se evalúan las expresiones de Lisp. Una ventaja de expresar el evaluador como un programa es que podemos ejecutar el programa. Esto nos da, ejecutándose dentro de Lisp, un modelo funcional de cómo Lisp mismo evalúa las expresiones. Esto puede servir como un marco para experimentar con las reglas de evaluación, como haremos más adelante en este capítulo.

Nuestro programa evaluador reduce las expresiones en última instancia a la aplicación de procedimientos primitivos. Por lo tanto, todo lo que necesitamos para ejecutar el evaluador es crear un mecanismo que llame al sistema Lisp subyacente para modelar la aplicación de los procedimientos primitivos.

Debe haber una ligadura para cada nombre de procedimiento primitivo, de modo que cuando `eval` evalúe el operador de una aplicación de una primitiva, encontrará un objeto para pasar a `apply`. Así, configuramos un entorno global que asocia objetos únicos con los nombres de los procedimientos primitivos que pueden aparecer en las expresiones que estaremos evaluando. El entorno global también incluye ligaduras para los símbolos `true` y `false`, de modo que puedan usarse como variables en las expresiones que se evaluarán.

(define (setup-environment) (let ((initial-env (extend-environment (primitive-procedure-names) (primitive-procedure-objects) the-empty-environment))) (define-variable! 'true true initial-env) (define-variable! 'false false initial-env) initial-env)) (define the-global-environment (setup-environment))

No importa cómo representemos los objetos de procedimiento primitivos, siempre que `apply` pueda identificarlos y aplicarlos usando los procedimientos `primitive/procedure?` y `apply/primitive/procedure`. Hemos elegido representar un procedimiento primitivo como una lista que comienza con el símbolo `primitive` y que contiene un procedimiento en el

Lisp subyacente que implementa esa primitiva.

(define (primitive-procedure? proc) (tagged-list? proc 'primitive)) (define (primitive-implementation proc) (cadr proc))

`setup/environment` obtendrá los nombres de las primitivas y los procedimientos de implementación de una lista:[16]

(define primitive-procedures (list (list 'car car) (list 'cdr cdr) (list 'cons cons) (list 'null? null?) < *más primitivas* >)) (define (primitive-procedure-names) (map car primitive-procedures)) (define (primitive-procedure-objects) (map (lambda (proc) (list 'primitive (cadr proc))) primitive-procedures))

Para aplicar un procedimiento primitivo, simplemente aplicamos el procedimiento de implementación a los argumentos, usando el sistema Lisp subyacente:[17]

(define (apply-primitive-procedure proc args) (apply-in-underlying-scheme (primitive-implementation proc) args))

Por conveniencia al ejecutar el evaluador metacircular, proporcionamos un bucle *driver* que modela el bucle de lectura-evaluación-impresión del sistema Lisp subyacente. Imprime un *prompt*, lee una expresión de entrada, evalúa esta expresión en el entorno global e imprime el resultado. Precedemos cada resultado impreso con un *prompt de salida* para distinguir el valor de la expresión de otras salidas que puedan imprimirse.[18]

(define input-prompt ";;; M-Eval input:") (define output-prompt ";;; M-Eval value:") (define (driver-loop) (prompt-for-input input-prompt) (let ((input (read))) (let ((output (eval input the-global-environment))) (announce-output output-prompt) (user-print output))) (driver-loop)) (define (prompt-for-input string) (newline) (newline) (display string) (newline)) (define (announce-output string) (newline) (display string) (newline))

Usamos un procedimiento de impresión especial, `user/print`, para evitar imprimir la parte de entorno de un procedimiento compuesto, que puede ser una lista muy larga (o incluso puede contener ciclos).

(define (user-print object) (if (compound-procedure? object) (display (list 'compound-procedure (procedure-parameters object) (procedure-body object) '<procedure-env>)) (display object)))

Ahora todo lo que necesitamos hacer para ejecutar el evaluador es inicializar el entorno global e iniciar el bucle del controlador. Aquí hay una interacción de ejemplo:

(define the-global-environment (setup-environment)) (driver-loop)

;;; M-Eval input: (define (append x y) (if (null? x) y (cons (car x) (append (cdr x) y)))) *;;; M-Eval value: ok ;;; M-Eval input:* (append '(a b c) '(d e f)) *;;; M-Eval value: (a b c d e f)*

> **Ejercicio 4.14:** Eva Lu Ator y Louis Reasoner están experimentando cada uno con el evaluador metacircular. Eva escribe la definición de `map` y ejecuta algunos programas de prueba que la usan. Funcionan bien. Louis, por el contrario, ha instalado la versión del sistema de `map` como una primitiva para el evaluador metacircular. Cuando lo prueba, las cosas van terriblemente mal. Explique por qué el `map` de Louis falla aunque el de Eva funciona.

Datos como Programas

Al pensar en un programa Lisp que evalúa expresiones Lisp, una analogía podría ser útil. Una visión operacional del significado de un programa es que un programa es una descripción de una máquina abstracta (quizás infinitamente grande). Por ejemplo, considere el programa familiar para calcular factoriales:

(define (factorial n) (if (= n 1) 1 (* (factorial (- n 1)) n)))

Podemos considerar este programa como la descripción de una máquina que contiene partes que decrementan, multiplican y prueban la igualdad, junto con un interruptor de dos posiciones y otra máquina factorial. (La máquina factorial es infinita porque contiene otra máquina factorial dentro de ella.) La Figura 4.2 es un diagrama de flujo para la máquina factorial, que muestra cómo están conectadas las partes.

De manera similar, podemos considerar al evaluador como una máquina muy especial que toma como entrada una descripción de una máquina. Dada esta entrada, el evaluador se configura a sí mismo para emular la máquina descrita. Por ejemplo, si le damos a nuestro evaluador la definición de `factorial`, como se muestra en la Figura 4.3, el evaluador podrá calcular factoriales.

Figura 4.2: El programa factorial, visto como una máquina abstracta.

Desde esta perspectiva, nuestro evaluador se ve como una *máquina universal*. Imita a otras máquinas cuando estas se describen como programas de Lisp.

Esto es sorprendente. Intente imaginar un evaluador análogo para circuitos eléctricos. Este sería un circuito que toma como entrada una señal que codifica los planos de algún otro circuito, como un filtro. Dada esta entrada, el evaluador del circuito se comportaría entonces como un filtro con la misma descripción. Tal circuito eléctrico universal es casi inimaginablemente complejo. Es notable que el evaluador de programas sea un programa bastante simple.[20]

Figura 4.3: El evaluador emulando una máquina factorial.

Otro aspecto sorprendente del evaluador es que actúa como un puente entre los objetos de datos que son manipulados por nuestro lenguaje de programación y el lenguaje de programación en sí. Imagine que el programa evaluador (implementado en Lisp) se está ejecutando, y que un usuario está escribiendo expresiones al evaluador y observando los resultados. Desde la perspectiva del usuario, una expresión de entrada como `(* x x)` es una expresión en el lenguaje de programación, que el evaluador debe ejecutar. Desde la perspectiva del evaluador, sin embargo, la expresión es simplemente una lista (en este caso, una lista de tres símbolos: `*`, `x`, y `x`) que debe ser manipulada de acuerdo con un conjunto bien definido de reglas.

El hecho de que los programas del usuario sean los datos del evaluador no tiene por qué ser una fuente de confusión. De hecho, a veces es conveniente ignorar esta distinción, y dar al usuario la capacidad de evaluar explícitamente un objeto de datos como una expresión de Lisp, haciendo que `eval` esté disponible para su uso en los programas. Muchos dialectos de Lisp proporcionan un procedimiento primitivo `eval` que toma como argumentos una expresión y un entorno y evalúa la expresión en relación con el entorno.[21] Por lo tanto,

```
(eval '(* 5 5) user-initial-environment)
```

y

```
(eval (cons '* (list 5 5)) user-initial-environment)
```

ambos devolverán 25.[22]

> **Ejercicio 4.15:** Dado un procedimiento de un argumento `p` y un objeto `a`, se dice que `p` se "detiene" en `a` si evaluar la expresión `(p a)` devuelve un valor (en lugar de terminar con un mensaje de error o ejecutarse para siempre). Muestre que es imposible escribir un procedimiento `halts?` que determine correctamente si `p` se detiene en `a` para cualquier

procedimiento `p` y objeto `a`. Use el siguiente razonamiento: si tuviera tal procedimiento `halts?`, podría implementar el siguiente programa:

(define (run-forever) (run-forever)) (define (try p) (if (halts? p p) (run-forever) 'halted))

Ahora considere evaluar la expresión `(try try)` y muestre que cualquier resultado posible (ya sea detenerse o ejecutarse para siempre) viola el comportamiento previsto de `halts?`.[23]

Definiciones Internas

Nuestro modelo de evaluación de entorno y nuestro evaluador metacircular ejecutan las definiciones en secuencia, extendiendo el marco del entorno una definición a la vez. Esto es particularmente conveniente para el desarrollo interactivo de programas, en el que el programador necesita mezclar libremente la aplicación de procedimientos con la definición de nuevos procedimientos. Sin embargo, si pensamos cuidadosamente en las definiciones internas utilizadas para implementar la estructura de bloques (introducida en Section 1.1.8), descubriremos que la extensión nombre por nombre del entorno puede no ser la mejor manera de definir las variables locales.

Considere un procedimiento con definiciones internas, como

(define (f x) (define (even? n) (if (= n 0) true (odd? (- n 1)))) (define (odd? n) (if (= n 0) false (even? (- n 1)))) ⟨ *resto del cuerpo de* `f` ⟩)

Nuestra intención aquí es que el nombre `odd?` en el cuerpo del procedimiento `even?` se refiera al procedimiento `odd?` que se define después de `even?`. El alcance del nombre `odd?` es todo el cuerpo de `f`, no solo la porción del cuerpo de `f` que comienza en el punto donde ocurre la `define` para `odd?`. De hecho, cuando consideramos que `odd?` se define a su vez en términos de `even?`—de modo que `even?` y `odd?` son procedimientos mutuamente recursivos—vemos que la única interpretación satisfactoria de las dos `define`s es considerarlas como si los nombres `even?` y `odd?` se estuvieran agregando al entorno simultáneamente. Más generalmente, en la estructura de bloques, el alcance de un nombre local es todo el cuerpo del procedimiento en el que se evalúa la `define`.

Resulta que nuestro intérprete evaluará correctamente las llamadas a `f`, pero por una razón "accidental": como las definiciones de los procedimientos internos vienen primero, no se evaluará ninguna llamada a estos procedimientos hasta que todos ellos hayan sido definidos. Por lo tanto, `odd?`

habrá sido definido para cuando se ejecute `even?`. De hecho, nuestro mecanismo de evaluación secuencial dará el mismo resultado que un mecanismo que implementa directamente la definición simultánea para cualquier procedimiento en el que las definiciones internas vengan primero en un cuerpo y la evaluación de las expresiones de valor para las variables definidas no use realmente ninguna de las variables definidas. (Para un ejemplo de un procedimiento que no obedece estas restricciones, de modo que la definición secuencial no es equivalente a la definición simultánea, vea el Ejercicio 4.19.)[24]

Hay, sin embargo, una forma sencilla de tratar las definiciones para que los nombres definidos internamente tengan un alcance verdaderamente simultáneo: simplemente crear todas las variables locales que estarán en el entorno actual antes de evaluar cualquiera de las expresiones de valor. Una forma de hacer esto es mediante una transformación de sintaxis en las expresiones `lambda`. Antes de evaluar el cuerpo de una expresión `lambda`, "escaneamos" y eliminamos todas las definiciones internas del cuerpo. Las variables definidas internamente se crearán con un `let` y luego se les asignarán sus valores mediante asignación. Por ejemplo, el procedimiento

(lambda < *vars* > (define u < *e1* >) (define v < *e2* >) < *e3* >)

se transformaría en

(lambda < *vars* > (let ((u '*unassigned*) (v '*unassigned*)) (set! u < *e1* >) (set! v < *e2* >) < *e3* >))

donde `*unassigned*` es un símbolo especial que hace que la búsqueda de una variable señale un error si se intenta usar el valor de la variable aún no asignada.

Una estrategia alternativa para escanear las definiciones internas se muestra en el Ejercicio 4.18. A diferencia de la transformación mostrada anteriormente, esta impone la restricción de que los valores de las variables definidas pueden evaluarse sin usar ninguno de los valores de las variables.[25]

> **Ejercicio 4.16:** En este ejercicio implementamos el método recién descrito para interpretar las definiciones internas. Asumimos que el evaluador soporta `let` (ver Ejercicio 4.6).
>
> 1. Cambie `lookup/variable/value` (Section 4.1.3) para que señale un error si el valor que encuentra es el símbolo `*unassigned*`.
>
> 2. Escriba un procedimiento `scan/out/defines` que tome un cuerpo de procedimiento y devuelva uno equivalente que no tenga definiciones

internas, haciendo la transformación descrita anteriormente.

3. Instale `scan/out/defines` en el intérprete, ya sea en `make/procedure` o en `procedure/body` (ver Section 4.1.3). ¿Qué lugar es mejor? ¿Por qué?

Ejercicio 4.17: Dibuje diagramas del entorno vigente al evaluar la expresión ⟨*e3*$\kern0.1em\rangle$ en el procedimiento del texto, comparando cómo se estructurará cuando las definiciones se interpreten secuencialmente con cómo se estructurará si las definiciones se escanean como se describe. ¿Por qué hay un marco adicional en el programa transformado? Explique por qué esta diferencia en la estructura del entorno nunca puede marcar una diferencia en el comportamiento de un programa correcto. Diseñe una forma de hacer que el intérprete implemente la regla de alcance "simultáneo" para las definiciones internas sin construir el marco adicional.

Ejercicio 4.18: Considere una estrategia alternativa para escanear las definiciones que traduce el ejemplo del texto a

(lambda ⟨ *vars* ⟩ (let ((u '*unassigned*) (v '*unassigned*)) (let ((a ⟨ *e1* ⟩) (b ⟨ *e2* ⟩)) (set! u a) (set! v b)) ⟨ *e3* ⟩))

Aquí `a` y `b` están destinados a representar nuevos nombres de variables, creados por el intérprete, que no aparecen en el programa del usuario. Considere el procedimiento `solve` de Section 3.5.4:

(define (solve f y0 dt) (define y (integral (delay dy) y0 dt)) (define dy (stream-map f y)) y)

¿Funcionará este procedimiento si las definiciones internas se escanean como se muestra en este ejercicio? ¿Y si se escanean como se muestra en el texto? Explique.

Ejercicio 4.19: Ben Bitdiddle, Alyssa P. Hacker y Eva Lu Ator están discutiendo sobre el resultado deseado de evaluar la expresión

(let ((a 1)) (define (f x) (define b (+ a x)) (define a 5) (+ a b)) (f 10))

Ben afirma que el resultado debería obtenerse usando la regla secuencial para `define`: `b` se define como 11, luego `a` se define como 5, por lo que el resultado es 16. Alyssa objeta que la recursión mutua requiere la regla de ámbito simultáneo para las definiciones de procedimientos internos, y que no es razonable tratar los nombres de los procedimientos de manera diferente a otros nombres. Por lo tanto, ella aboga por el mecanismo

implementado en el Ejercicio 4.16. Esto llevaría a que `a` no estuviera asignada en el momento en que se debe calcular el valor de `b`. Por lo tanto, en la opinión de Alyssa, el procedimiento debería producir un error. Eva tiene una tercera opinión. Ella dice que si las definiciones de `a` y `b` están verdaderamente destinadas a ser simultáneas, entonces el valor 5 para `a` debería usarse al evaluar `b`. Por lo tanto, en la opinión de Eva, `a` debería ser 5, `b` debería ser 15 y el resultado debería ser 20. ¿Cuál (si es que hay alguna) de estas opiniones apoya usted? ¿Puede idear una forma de implementar las definiciones internas para que se comporten como prefiere Eva?[26]

Ejercicio 4.20: Debido a que las definiciones internas parecen secuenciales pero en realidad son simultáneas, algunas personas prefieren evitarlas por completo y usar en su lugar la forma especial `letrec`. `letrec` se parece a `let`, por lo que no es sorprendente que las variables que liga estén ligadas simultáneamente y tengan el mismo alcance entre sí. El procedimiento de ejemplo `f` anterior puede escribirse sin definiciones internas, pero con exactamente el mismo significado, como

(define (f x) (letrec ((even? (lambda (n) (if (= n 0) true (odd? (- n 1))))) (odd? (lambda (n) (if (= n 0) false (even? (- n 1)))))) ⟨ *resto del cuerpo de* `f` ⟩))

Las expresiones `letrec`, que tienen la forma

(letrec ((⟨ *var* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$ ⟨ *exp* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$) … (⟨ *var* $\color{SchemeDark}_{\hbox{\ttfamily \itshape\scriptsize n}}\rangle$ ⟨ *exp* $\color{SchemeDark} _{\hbox{\ttfamily\itshape\scriptsize n}}\rangle$)) ⟨ *cuerpo* ⟩)

son una variación de `let` en la que las expresiones ⟨*expk*⟩ que proporcionan los valores iniciales para las variables ⟨*vark*⟩ se evalúan en un entorno que incluye todas las ligaduras de `letrec`. Esto permite la recursión en las ligaduras, como la recursión mutua de `even?` y `odd?` en el ejemplo anterior, o la evaluación de 10 factorial con

(letrec ((fact (lambda (n) (if (= n 1) 1 (* n (fact (- n 1))))))) (fact 10))

1. Implemente `letrec` como una expresión derivada, transformando una expresión `letrec` en una expresión `let` como se muestra en el texto anterior o en el Ejercicio 4.18. Es decir, las variables `letrec` deberían crearse con un `let` y luego asignarles sus valores con `set!`.

2. Louis Reasoner está confundido por todo este alboroto sobre las

definiciones internas. A su modo de ver, si no te gusta usar `define` dentro de un procedimiento, puedes usar `let`. Ilustre lo que está flojo en su razonamiento dibujando un diagrama de entorno que muestre el entorno en el que se evalúa el ⟨*resto del cuerpo de* `f`$\kern0.08em \rangle$ durante la evaluación de la expresión `(f 5)`, con `f` definido como en este ejercicio. Dibuje un diagrama de entorno para la misma evaluación, pero con `let` en lugar de `letrec` en la definición de `f`.

Ejercicio 4.21: Sorprendentemente, la intuición de Louis en el Ejercicio 4.20 es correcta. De hecho, es posible especificar procedimientos recursivos sin usar `letrec` (o incluso `define`), aunque el método para lograr esto es mucho más sutil de lo que Louis imaginó. La siguiente expresión calcula 10 factorial aplicando un procedimiento factorial recursivo:[27]

((lambda (n) ((lambda (fact) (fact fact n)) (lambda (ft k) (if (= k 1) 1 (* k (ft ft (- k 1))))))) 10)

1. Comprueba (evaluando la expresión) que esto realmente calcula factoriales. Diseña una expresión análoga para calcular números de Fibonacci.

2. Considere el siguiente procedimiento, que incluye definiciones internas mutuamente recursivas:

   (define (f x) (define (even? n) (if (= n 0) true (odd? (- n 1)))) (define (odd? n) (if (= n 0) false (even? (- n 1)))) (even? x))

   Rellena las expresiones que faltan para completar una definición alternativa de `f`, que no utiliza ni definiciones internas ni `letrec`:

   (define (f x) ((lambda (even? odd?) (even? even? odd? x)) (lambda (ev? od? n) (if (= n 0) true (od? ⟨??⟩ ⟨??⟩ ⟨??⟩))) (lambda (ev? od? n) (if (= n 0) false (ev? ⟨??⟩ ⟨??⟩ ⟨??⟩)))))

Separando el Análisis Sintáctico de la Ejecución

El evaluador implementado anteriormente es simple, pero es muy ineficiente, porque el análisis sintáctico de las expresiones se entrelaza con su ejecución. Por lo tanto, si un programa se ejecuta muchas veces, su sintaxis se analiza muchas veces. Considere, por ejemplo, evaluar `(factorial 4)` usando la siguiente definición de `factorial`:

(define (factorial n) (if (= n 1) 1 (* (factorial (- n 1)) n)))

Cada vez que se llama a `factorial`, el evaluador debe determinar que el cuerpo es una expresión `if` y extraer el predicado. Solo entonces puede evaluar el predicado y despachar según su valor. Cada vez que evalúa la expresión `(* (factorial (- n 1)) n)`, o las subexpresiones `(factorial (- n 1))` y `(- n 1)`, el evaluador debe realizar el análisis de casos en `eval` para determinar que la expresión es una aplicación, y debe extraer su operador y operandos. Este análisis es costoso. Realizarlo repetidamente es un desperdicio. Podemos transformar el evaluador para que sea significativamente más eficiente organizando las cosas de modo que el análisis sintáctico se realice solo una vez.[28] Dividimos `eval`, que toma una expresión y un entorno, en dos partes. El procedimiento `analyze` toma solo la expresión. Realiza el análisis sintáctico y devuelve un nuevo procedimiento, el *procedimiento de ejecución*, que encapsula el trabajo a realizar en la ejecución de la expresión analizada. El procedimiento de ejecución toma un entorno como argumento y completa la evaluación. Esto ahorra trabajo porque `analyze` se llamará solo una vez en una expresión, mientras que el procedimiento de ejecución puede llamarse muchas veces. Con la separación en análisis y ejecución, `eval` ahora se convierte en

```
(define (eval exp env) ((analyze exp) env))
```

El resultado de llamar a `analyze` es el procedimiento de ejecución que se aplicará al entorno. El procedimiento `analyze` es el mismo análisis de casos que el realizado por el `eval` original de Section 4.1.1, excepto que los procedimientos a los que despachamos realizan solo análisis, no la evaluación completa:

```
(define (analyze exp) (cond ((self-evaluating? exp) (analyze-self-evaluating exp)) ((quoted? exp) (analyze-quoted exp)) ((variable? exp) (analyze-variable exp)) ((assignment? exp) (analyze-assignment exp)) ((definition? exp) (analyze-definition exp)) ((if? exp) (analyze-if exp)) ((lambda? exp) (analyze-lambda exp)) ((begin? exp) (analyze-sequence (begin-actions exp))) ((cond? exp) (analyze (cond->if exp))) ((application? exp) (analyze-application exp)) (else (error "Unknown expression type: ANALYZE" exp))))
```

Aquí está el procedimiento de análisis sintáctico más simple, que maneja las expresiones que se autoevalúan. Devuelve un procedimiento de ejecución que ignora su argumento de entorno y simplemente devuelve la expresión:

```
(define (analyze-self-evaluating exp) (lambda (env) exp))
```

Para una expresión entrecomillada, podemos ganar un poco de eficiencia extrayendo el texto de la cita solo una vez, en la fase de análisis, en lugar de en la fase de ejecución.

(define (analyze-quoted exp) (let ((qval (text-of-quotation exp))) (lambda (env) qval)))

La búsqueda del valor de una variable todavía debe hacerse en la fase de ejecución, ya que esto depende de conocer el entorno.[29]

(define (analyze-variable exp) (lambda (env) (lookup-variable-value exp env)))

`analyze/assignment` también debe posponer el establecimiento real de la variable hasta la ejecución, cuando se haya proporcionado el entorno. Sin embargo, el hecho de que la expresión `assignment/value` pueda analizarse (recursivamente) durante el análisis es una ganancia importante en eficiencia, porque la expresión `assignment/value` ahora se analizará solo una vez. Lo mismo ocurre con las definiciones.

(define (analyze-assignment exp) (let ((var (assignment-variable exp)) (vproc (analyze (assignment-value exp)))) (lambda (env) (set-variable-value! var (vproc env) env) 'ok))) (define (analyze-definition exp) (let ((var (definition-variable exp)) (vproc (analyze (definition-value exp)))) (lambda (env) (define-variable! var (vproc env) env) 'ok)))

Para las expresiones `if`, extraemos y analizamos el predicado, el consecuente y la alternativa en el momento del análisis.

(define (analyze-if exp) (let ((pproc (analyze (if-predicate exp))) (cproc (analyze (if-consequent exp))) (aproc (analyze (if-alternative exp)))) (lambda (env) (if (true? (pproc env)) (cproc env) (aproc env)))))

Analizar una expresión `lambda` también logra una ganancia importante en eficiencia: analizamos el cuerpo de `lambda` solo una vez, aunque los procedimientos resultantes de la evaluación de la `lambda` pueden aplicarse muchas veces.

(define (analyze-lambda exp) (let ((vars (lambda-parameters exp)) (bproc (analyze-sequence (lambda-body exp)))) (lambda (env) (make-procedure vars bproc env))))

El análisis de una secuencia de expresiones (como en un `begin` o el cuerpo de una expresión `lambda`) es más complicado.[30] Cada expresión en la secuencia se analiza, produciendo un procedimiento de ejecución. Estos procedimientos de ejecución se combinan para producir un procedimiento de ejecución que toma un entorno como argumento y llama secuencialmente a cada procedimiento de ejecución individual con el entorno como argumento.

(define (analyze-sequence exps) (define (sequentially proc1 proc2) (lambda (env) (proc1 env) (proc2 env))) (define (loop first-proc rest-procs) (if (null? rest-procs) first-proc (loop (sequentially first-proc (car rest-procs)) (cdr rest-procs)))) (let ((procs (map analyze exps))) (if (null? procs) (error "Empty sequence: ANALYZE")) (loop (car procs) (cdr procs))))

Para analizar una aplicación, analizamos el operador y los operandos y construimos un procedimiento de ejecución que llama al procedimiento de ejecución del operador (para obtener el procedimiento real que se aplicará) y a los procedimientos de ejecución de los operandos (para obtener los argumentos reales). Luego los pasamos a `execute/application`, que es el análogo de `apply` en Section 4.1.1. `execute/application` difiere de `apply` en que el cuerpo del procedimiento para un procedimiento compuesto ya ha sido analizado, por lo que no hay necesidad de hacer más análisis. En su lugar, simplemente llamamos al procedimiento de ejecución para el cuerpo en el entorno extendido.

(define (analyze-application exp) (let ((fproc (analyze (operator exp))) (aprocs (map analyze (operands exp)))) (lambda (env) (execute-application (fproc env) (map (lambda (aproc) (aproc env)) aprocs))))) (define (execute-application proc args) (cond ((primitive-procedure? proc) (apply-primitive-procedure proc args)) ((compound-procedure? proc) ((procedure-body proc) (extend-environment (procedure-parameters proc) args (procedure-environment proc)))) (else (error "Unknown procedure type: EXECUTE-APPLICATION" proc))))

Nuestro nuevo evaluador utiliza las mismas estructuras de datos, procedimientos de sintaxis y procedimientos de soporte en tiempo de ejecución que en las secciones Sección 4.1.2, Sección 4.1.3 y Sección 4.1.4.

> **Ejercicio 4.22:** Extienda el evaluador de esta sección para que admita la forma especial `let`. (Ver Ejercicio 4.6.)
>
> **Ejercicio 4.23:** Alyssa P. Hacker no entiende por qué `analyze/sequence` necesita ser tan complicado. Todos los demás procedimientos de análisis son transformaciones directas de los procedimientos de evaluación correspondientes (o cláusulas `eval`) en Section 4.1.1. Ella esperaba que `analyze/sequence` se viera así:
>
> (define (analyze-sequence exps) (define (execute-sequence procs env) (cond ((null? (cdr procs)) ((car procs) env)) (else ((car procs) env) (execute-sequence (cdr procs) env)))) (let ((procs (map analyze exps))) (if (null? procs) (error "Empty sequence: ANALYZE")) (lambda (env) (execute-sequence procs env))))

Eva Lu Ator le explica a Alyssa que la versión del texto hace más trabajo de evaluación de una secuencia en el momento del análisis. El procedimiento de ejecución de secuencias de Alyssa, en lugar de tener las llamadas a los procedimientos de ejecución individuales incorporadas, recorre los procedimientos para llamarlos: en efecto, aunque las expresiones individuales de la secuencia han sido analizadas, la secuencia en sí no lo ha sido. Compare las dos versiones de `analyze/sequence`. Por ejemplo, considere el caso común (típico de los cuerpos de los procedimientos) donde la secuencia tiene una sola expresión. ¿Qué trabajo hará el procedimiento de ejecución producido por el programa de Alyssa? ¿Y el procedimiento de ejecución producido por el programa del texto anterior? ¿Cómo se comparan las dos versiones para una secuencia con dos expresiones?

Ejercicio 4.24: Diseñe y lleve a cabo algunos experimentos para comparar la velocidad del evaluador metacircular original con la versión de esta sección. Use sus resultados para estimar la fracción de tiempo que se dedica al análisis frente a la ejecución para varios procedimientos.

Variaciones sobre un Esquema — Evaluación Perezosa

Ahora que tenemos un evaluador expresado como un programa Lisp, podemos experimentar con opciones alternativas en el diseño de lenguajes simplemente modificando el evaluador. De hecho, a menudo se inventan nuevos lenguajes escribiendo primero un evaluador que incrusta el nuevo lenguaje dentro de un lenguaje de alto nivel existente. Por ejemplo, si deseamos discutir algún aspecto de una modificación propuesta a Lisp con otro miembro de la comunidad Lisp, podemos proporcionar un evaluador que encarne el cambio. El destinatario puede entonces experimentar con el nuevo evaluador y enviar comentarios como modificaciones adicionales. No solo la base de implementación de alto nivel facilita la prueba y depuración del evaluador; además, la incrustación permite al diseñador tomar prestadas[31] características del lenguaje subyacente, al igual que nuestro evaluador de Lisp incrustado usa primitivas y estructuras de control del Lisp subyacente. Solo más tarde (si es que alguna vez) el diseñador necesita tomarse la molestia de construir una implementación completa en un lenguaje de bajo nivel o en hardware. En esta sección y la siguiente exploraremos algunas variaciones de Scheme que proporcionan un poder expresivo adicional significativo.

Orden Normal y Orden Aplicativo

En Section 1.1, donde comenzamos nuestra discusión sobre los modelos de evaluación, notamos que Scheme es un lenguaje de *orden aplicativo*, es decir, que todos los argumentos de los procedimientos de Scheme se evalúan cuando se aplica el procedimiento. En contraste, los lenguajes de *orden normal* retrasan la evaluación de los argumentos de los procedimientos hasta que se necesitan los valores reales de los argumentos. Retrasar la evaluación de los argumentos de los procedimientos hasta el último momento posible (por ejemplo, hasta que sean requeridos por una operación primitiva) se llama *evaluación perezosa*.[32] Considere el procedimiento

(define (try a b) (if (= a 0) 1 b))

Evaluar `(try 0 (/ 1 0))` genera un error en Scheme. Con la evaluación perezosa, no habría ningún error. Evaluar la expresión devolvería 1, porque el argumento `(/ 1 0)` nunca se evaluaría. Un ejemplo que explota la evaluación perezosa es la definición de un procedimiento `unless`

(define (unless condition usual-value exceptional-value) (if condition exceptional-value usual-value))

que se puede usar en expresiones como

(unless (= b 0) (/ a b) (begin (display "exception: returning 0") 0))

Esto no funcionará en un lenguaje de orden aplicativo porque tanto el valor usual como el valor excepcional se evaluarán antes de que se llame a `unless` (compare el Ejercicio 1.6). Una ventaja de la evaluación perezosa es que algunos procedimientos, como `unless`, pueden realizar cálculos útiles incluso si la evaluación de algunos de sus argumentos produciría errores o no terminaría. Si el cuerpo de un procedimiento se introduce antes de que un argumento haya sido evaluado, decimos que el procedimiento es *no estricto* en ese argumento. Si el argumento se evalúa antes de que se introduzca el cuerpo del procedimiento, decimos que el procedimiento es *estricto* en ese argumento.[33] En un lenguaje puramente de orden aplicativo, todos los procedimientos son estrictos en cada argumento. En un lenguaje puramente de orden normal, todos los procedimientos compuestos son no estrictos en cada argumento, y los procedimientos primitivos pueden ser estrictos o no estrictos. También hay lenguajes (ver Ejercicio 4.31) que dan a los programadores un control detallado sobre la estrictez de los procedimientos que definen. Un ejemplo sorprendente de un procedimiento que puede hacerse útilmente no estricto es `cons` (o, en general, casi cualquier constructor para estructuras de datos). Se pueden realizar cálculos útiles,

combinando elementos para formar estructuras de datos y operando sobre las estructuras de datos resultantes, incluso si los valores de los elementos no se conocen. Tiene perfecto sentido, por ejemplo, calcular la longitud de una lista sin conocer los valores de los elementos individuales de la lista. Explotaremos esta idea en Section 4.2.3 para implementar los flujos del Capítulo 3 como listas formadas por pares `cons` no estrictos.

Ejercicio 4.25: Supongamos que (en Scheme de orden aplicativo ordinario) definimos `unless` como se muestra arriba y luego definimos `factorial` en términos de `unless` como

(define (factorial n) (unless (= n 1) (* n (factorial (- n 1))) 1))

¿Qué sucede si intentamos evaluar `(factorial 5)`? ¿Funcionarán nuestras definiciones en un lenguaje de orden normal?

Ejercicio 4.26: Ben Bitdiddle y Alyssa P. Hacker no están de acuerdo sobre la importancia de la evaluación perezosa para implementar cosas como `unless`. Ben señala que es posible implementar `unless` en orden aplicativo como una forma especial. Alyssa replica que, si se hiciera eso, `unless` sería simplemente sintaxis, no un procedimiento que podría usarse junto con procedimientos de orden superior. Complete los detalles de ambos lados del argumento. Muestre cómo implementar `unless` como una expresión derivada (como `cond` o `let`), y dé un ejemplo de una situación en la que podría ser útil tener `unless` disponible como un procedimiento, en lugar de como una forma especial.

Un Intérprete con Evaluación Perezosa

En esta sección implementaremos un lenguaje de orden normal que es igual a Scheme excepto que los procedimientos compuestos no son estrictos en cada argumento. Los procedimientos primitivos seguirán siendo estrictos. No es difícil modificar el evaluador de Section 4.1.1 para que el lenguaje que interpreta se comporte de esta manera. Casi todos los cambios requeridos se centran en la aplicación de procedimientos. La idea básica es que, al aplicar un procedimiento, el intérprete debe determinar qué argumentos deben evaluarse y cuáles deben retrasarse. Los argumentos retrasados no se evalúan; en su lugar, se transforman en objetos llamados *thunks*.[34] El thunk debe contener la información requerida para producir el valor del argumento cuando sea necesario, como si hubiera sido evaluado en el momento de la aplicación. Por lo tanto, el thunk debe contener la expresión del argumento y el entorno en el que se está evaluando la aplicación del procedimiento. El proceso de evaluar la expresión en un thunk se llama *forzar*.[35] En general, un thunk se forzará solo cuando se necesite su valor: cuando se pase a un

procedimiento primitivo que usará el valor del thunk; cuando sea el valor de un predicado de un condicional; y cuando sea el valor de un operador que está a punto de aplicarse como un procedimiento. Una opción de diseño que tenemos disponible es si *memoizar* o no los thunks, como hicimos con los objetos retardados en Section 3.5.1. Con la memoización, la primera vez que se fuerza un thunk, almacena el valor que se calcula. Las forzadas posteriores simplemente devuelven el valor almacenado sin repetir el cálculo. Haremos que nuestro intérprete memoize, porque esto es más eficiente para muchas aplicaciones. Aquí, sin embargo, hay consideraciones delicadas.[36]

Modificando el evaluador

La principal diferencia entre el evaluador perezoso y el de Section 4.1 está en el manejo de las aplicaciones de procedimientos en `eval` y `apply`. La cláusula `application?` de `eval` se convierte en

((application? exp) (apply (actual-value (operator exp) env) (operands exp) env))

Esto es casi lo mismo que la cláusula `application?` de `eval` en Section 4.1.1. Sin embargo, para la evaluación perezosa, llamamos a `apply` con las expresiones de los operandos, en lugar de los argumentos producidos al evaluarlos. Como necesitaremos el entorno para construir thunks si los argumentos deben retrasarse, también debemos pasarlo. Todavía evaluamos el operador, porque `apply` necesita el procedimiento real que se aplicará para despachar según su tipo (primitivo versus compuesto) y aplicarlo. Siempre que necesitemos el valor real de una expresión, usamos

(define (actual-value exp env) (force-it (eval exp env)))

en lugar de solo `eval`, para que si el valor de la expresión es un thunk, se fuerce. Nuestra nueva versión de `apply` también es casi la misma que la versión de la Sección 4.1.1. La diferencia es que `eval` ha pasado expresiones de operandos no evaluadas: para los procedimientos primitivos (que son estrictos), evaluamos todos los argumentos antes de aplicar la primitiva; para los procedimientos compuestos (que no son estrictos), retrasamos todos los argumentos antes de aplicar el procedimiento.

(define (apply procedure arguments env) (cond ((primitive-procedure? procedure) (apply-primitive-procedure procedure (list-of-arg-values arguments env))) ; cambiado ((compound-procedure? procedure) (eval-sequence (procedure-body procedure) (extend-environment (procedure-parameters procedure) (list-of-delayed-args arguments env) ; cambiado (procedure-environment procedure)))) (else (error "Unknown procedure type:

APPLY" procedure))))

Los procedimientos que procesan los argumentos son como `list/of/values` de Section 4.1.1, excepto que `list/of/delayed/args` retrasa los argumentos en lugar de evaluarlos, y `list/of/arg/values` usa `actual/value` en lugar de `eval`:

(define (list-of-arg-values exps env) (if (no-operands? exps) '() (cons (actual-value (first-operand exps) env) (list-of-arg-values (rest-operands exps) env)))) (define (list-of-delayed-args exps env) (if (no-operands? exps) '() (cons (delay-it (first-operand exps) env) (list-of-delayed-args (rest-operands exps) env))))

El otro lugar donde debemos cambiar el evaluador es en el manejo de `if`, donde debemos usar `actual/value` en lugar de `eval` para obtener el valor de la expresión del predicado antes de probar si es verdadero o falso:

(define (eval-if exp env) (if (true? (actual-value (if-predicate exp) env)) (eval (if-consequent exp) env) (eval (if-alternative exp) env)))

Finalmente, debemos cambiar el procedimiento `driver/loop` (Section 4.1.4) para usar `actual/value` en lugar de `eval`, de modo que si un valor retardado se propaga de vuelta al bucle de lectura-evaluación-impresión, se forzará antes de ser impreso. También cambiamos los avisos para indicar que este es el evaluador perezoso:

(define input-prompt ";;; L-Eval input:") (define output-prompt ";;; L-Eval value:") (define (driver-loop) (prompt-for-input input-prompt) (let ((input (read))) (let ((output (actual-value input the-global-environment))) (announce-output output-prompt) (user-print output))) (driver-loop))

Con estos cambios hechos, podemos iniciar el evaluador y probarlo. La evaluación exitosa de la expresión `try` discutida en Section 4.2.1 indica que el intérprete está realizando una evaluación perezosa:

(define the-global-environment (setup-environment)) (driver-loop) *;;; L-Eval input:* (define (try a b) (if (= a 0) 1 b)) *;;; L-Eval value: ok ;;; L-Eval input:* (try 0 (/ 1 0)) *;;; L-Eval value: 1*

Representando thunks

Nuestro evaluador debe encargarse de crear thunks cuando se aplican procedimientos a argumentos y de forzar estos thunks más tarde. Un thunk debe empaquetar una expresión junto con el entorno, para que el argumento pueda producirse más tarde. Para forzar el thunk, simplemente extraemos la

expresión y el entorno del thunk y evaluamos la expresión en el entorno. Usamos `actual/value` en lugar de `eval` para que, en caso de que el valor de la expresión sea a su vez un thunk, forcemos ese, y así sucesivamente, hasta que lleguemos a algo que no sea un thunk:

(define (force-it obj) (if (thunk? obj) (actual-value (thunk-exp obj) (thunk-env obj)) obj))

Una forma fácil de empaquetar una expresión con un entorno es hacer una lista que contenga la expresión y el entorno. Por lo tanto, creamos un thunk de la siguiente manera:

(define (delay-it exp env) (list 'thunk exp env)) (define (thunk? obj) (tagged-list? obj 'thunk)) (define (thunk-exp thunk) (cadr thunk)) (define (thunk-env thunk) (caddr thunk))

En realidad, lo que queremos para nuestro intérprete no es exactamente esto, sino más bien thunks que han sido memoizados. Cuando se fuerza un thunk, lo convertiremos en un thunk evaluado reemplazando la expresión almacenada con su valor y cambiando la etiqueta `thunk` para que pueda reconocerse como ya evaluado.[37]

(define (evaluated-thunk? obj) (tagged-list? obj 'evaluated-thunk)) (define (thunk-value evaluated-thunk) (cadr evaluated-thunk)) (define (force-it obj) (cond ((thunk? obj) (let ((result (actual-value (thunk-exp obj) (thunk-env obj)))) (set-car! obj 'evaluated-thunk) (set-car! (cdr obj) result) ; reemplazar `exp` con su valor (set-cdr! (cdr obj) '()) ; olvidar el `env` innecesario result)) ((evaluated-thunk? obj) (thunk-value obj)) (else obj)))

Nótese que el mismo procedimiento `delay/it` funciona tanto con como sin memoización.

Ejercicio 4.27: Supongamos que escribimos las siguientes definiciones en el evaluador perezoso:

(define count 0) (define (id x) (set! count (+ count 1)) x)

Dé los valores que faltan en la siguiente secuencia de interacciones y explique sus respuestas.[38]

(define w (id (id 10))) *;;; L-Eval input:* count *;;; L-Eval value:* *< respuesta >* *;;; L-Eval input:* w *;;; L-Eval value:* *< respuesta >* *;;; L-Eval input:* count *;;; L-Eval value:* *< respuesta >*

Ejercicio 4.28: `eval` usa `actual/value` en lugar de `eval` para evaluar el

operador antes de pasarlo a `apply`, con el fin de forzar el valor del operador. Dé un ejemplo que demuestre la necesidad de este forzamiento.
Ejercicio 4.29: Muestre un programa que esperaría que se ejecutara mucho más lentamente sin memoización que con memoización. Además, considere la siguiente interacción, donde el procedimiento `id` se define como en el Ejercicio 4.27 y `count` comienza en 0:

(define (square x) (* x x)) *;;; L-Eval input:* (square (id 10)) *;;; L-Eval value:* ⟨ *respuesta* ⟩ *;;; L-Eval input:* count *;;; L-Eval value:* ⟨ *respuesta* ⟩

Dé las respuestas tanto cuando el evaluador memoiza como cuando no lo hace.

Ejercicio 4.30: Cy D. Fect, un programador de C reformado, está preocupado de que algunos efectos secundarios nunca ocurran, porque el evaluador perezoso no fuerza las expresiones en una secuencia. Dado que el valor de una expresión en una secuencia que no sea la última no se usa (la expresión está ahí solo por su efecto, como asignar a una variable o imprimir), no puede haber un uso posterior de este valor (por ejemplo, como argumento de un procedimiento primitivo) que haga que se fuerce. Cy piensa, por lo tanto, que al evaluar secuencias, debemos forzar todas las expresiones de la secuencia excepto la final. Propone modificar `eval/sequence` de Section 4.1.1 para usar `actual/value` en lugar de `eval`:

(define (eval-sequence exps env) (cond ((last-exp? exps) (eval (first-exp exps) env)) (else (actual-value (first-exp exps) env) (eval-sequence (rest-exps exps) env))))

1. Ben Bitdiddle cree que Cy está equivocado. Le muestra a Cy el procedimiento `for/each` descrito en el Ejercicio 2.23, que da un ejemplo importante de una secuencia con efectos secundarios:

   (define (for-each proc items) (if (null? items) 'done (begin (proc (car items)) (for-each proc (cdr items)))))

   Afirma que el evaluador del texto (con el `eval/sequence` original) maneja esto correctamente:

   ;;; L-Eval input: (for-each (lambda (x) (newline) (display x)) (list 57 321 88)) *57 321 88 ;;; L-Eval value: done*

   Explique por qué Ben tiene razón sobre el comportamiento de `for/each`.

2. Cy está de acuerdo en que Ben tiene razón sobre el ejemplo de `for/each`, pero dice que ese no es el tipo de programa en el que estaba pensando cuando propuso su cambio a `eval/sequence`. Define los siguientes dos procedimientos en el evaluador perezoso:

   (define (p1 x) (set! x (cons x '(2))) x) (define (p2 x) (define (p e) e x) (p (set! x (cons x '(2)))))

   ¿Cuáles son los valores de `(p1 1)` y `(p2 1)` con el `eval/sequence` original? ¿Cuáles serían los valores con el cambio propuesto por Cy a `eval/sequence`?

3. Cy también señala que cambiar `eval/sequence` como él propone no afecta el comportamiento del ejemplo de la parte a. Explique por qué esto es cierto.

4. ¿Cómo cree que deberían tratarse las secuencias en el evaluador perezoso? ¿Le gusta el enfoque de Cy, el enfoque del texto o algún otro enfoque?

Ejercicio 4.31: El enfoque adoptado en esta sección es algo desagradable, porque introduce un cambio incompatible con Scheme. Podría ser más agradable implementar la evaluación perezosa como una extensión *compatible hacia arriba*, es decir, para que los programas ordinarios de Scheme funcionen como antes. Podemos hacer esto extendiendo la sintaxis de las declaraciones de procedimientos para permitir que el usuario controle si los argumentos deben ser retrasados o no. Ya que estamos en ello, también podríamos darle al usuario la opción de elegir entre retrasar con y sin memoización. Por ejemplo, la definición

(define (f a (b lazy) c (d lazy-memo)) …)

definiría `f` como un procedimiento de cuatro argumentos, donde el primer y tercer argumento se evalúan cuando se llama al procedimiento, el segundo argumento se retrasa, y el cuarto argumento se retrasa y se memoiza. Así, las definiciones de procedimientos ordinarios producirán el mismo comportamiento que el Scheme ordinario, mientras que agregar la declaración `lazy/memo` a cada parámetro de cada procedimiento compuesto producirá el comportamiento del evaluador perezoso definido en esta sección. Diseñe e implemente los cambios requeridos para producir tal extensión a Scheme. Tendrá que implementar nuevos procedimientos de sintaxis para manejar la nueva sintaxis para `define`. También debe hacer arreglos para que `eval` o `apply` determinen cuándo los argumentos deben ser retrasados, y para forzar o retrasar los argumentos en consecuencia, y debe hacer arreglos para que el

forzamiento memoice o no, según corresponda.

Flujos como Listas Perezosas

En Section 3.5.1, mostramos cómo implementar flujos como listas retardadas. Introdujimos las formas especiales `delay` y `cons/stream`, que nos permitieron construir una "promesa" de calcular el `cdr` de un flujo, sin cumplir realmente esa promesa hasta más tarde. Podríamos usar esta técnica general de introducir formas especiales cada vez que necesitemos más control sobre el proceso de evaluación, pero esto es incómodo. Por un lado, una forma especial no es un objeto de primera clase como un procedimiento, por lo que no podemos usarla junto con procedimientos de orden superior.[39] Además, nos vimos obligados a crear flujos como un nuevo tipo de objeto de datos similar pero no idéntico a las listas, y esto nos obligó a reimplementar muchas operaciones de lista ordinarias (`map`, `append`, y así sucesivamente) para su uso con flujos. Con la evaluación perezosa, los flujos y las listas pueden ser idénticos, por lo que no hay necesidad de formas especiales o de operaciones separadas para listas y flujos. Todo lo que necesitamos hacer es arreglar las cosas para que `cons` no sea estricto. Una forma de lograr esto es extender el evaluador perezoso para permitir primitivas no estrictas, e implementar `cons` como una de estas. Una forma más fácil es recordar (Section 2.1.3) que no hay una necesidad fundamental de implementar `cons` como una primitiva en absoluto. En su lugar, podemos representar los pares como procedimientos:[40]

(define (cons x y) (lambda (m) (m x y))) (define (car z) (z (lambda (p q) p))) (define (cdr z) (z (lambda (p q) q)))

En términos de estas operaciones básicas, las definiciones estándar de las operaciones de lista funcionarán tanto con listas infinitas (flujos) como con las finitas, y las operaciones de flujo pueden implementarse como operaciones de lista. Aquí hay algunos ejemplos:

(define (list-ref items n) (if (= n 0) (car items) (list-ref (cdr items) (- n 1)))) (define (map proc items) (if (null? items) '() (cons (proc (car items)) (map proc (cdr items))))) (define (scale-list items factor) (map (lambda (x) (* x factor)) items)) (define (add-lists list1 list2) (cond ((null? list1) list2) ((null? list2) list1) (else (cons (+ (car list1) (car list2)) (add-lists (cdr list1) (cdr list2)))))) (define ones (cons 1 ones)) (define integers (cons 1 (add-lists ones integers))) *;;; L-Eval input:* (list-ref integers 17) *;;; L-Eval value: 18*

Nótese que estas listas perezosas son aún más perezosas que los flujos de Capítulo 3: El `car` de la lista, así como el `cdr`, se retrasa.[41] De hecho, incluso acceder al `car` o al `cdr` de un par perezoso no necesita forzar el valor

de un elemento de la lista. El valor se forzará solo cuando sea realmente necesario, por ejemplo, para usarlo como argumento de una primitiva, o para imprimirlo como respuesta. Los pares perezosos también ayudan con el problema que surgió con los flujos en Section 3.5.4, donde descubrimos que la formulación de modelos de flujo de sistemas con bucles puede requerir que salpiquemos nuestros programas con operaciones explícitas de `delay`, más allá de las proporcionadas por `cons/stream`. Con la evaluación perezosa, todos los argumentos de los procedimientos se retrasan uniformemente. Por ejemplo, podemos implementar procedimientos para integrar listas y resolver ecuaciones diferenciales como pretendíamos originalmente en Section 3.5.4:

(define (integral integrand initial-value dt) (define int (cons initial-value (add-lists (scale-list integrand dt) int))) int) (define (solve f y0 dt) (define y (integral dy y0 dt)) (define dy (map f y)) y) *;;; L-Eval input:* (list-ref (solve (lambda (x) x) 1 0.001) 1000) *;;; L-Eval value: 2.716924*

Ejercicio 4.32: Dé algunos ejemplos que ilustren la diferencia entre los flujos del Capítulo 3 y las listas perezosas "más perezosas" descritas en esta sección. ¿Cómo puede aprovechar esta pereza adicional?

Ejercicio 4.33: Ben Bitdiddle prueba la implementación de la lista perezosa dada anteriormente evaluando la expresión:

(car '(a b c))

Para su sorpresa, esto produce un error. Después de pensarlo un poco, se da cuenta de que las "listas" obtenidas al leer expresiones entrecomilladas son diferentes de las listas manipuladas por las nuevas definiciones de `cons`, `car` y `cdr`. Modifique el tratamiento que el evaluador da a las expresiones entrecomilladas para que las listas entrecomilladas escritas en el bucle del controlador produzcan listas perezosas verdaderas.

Ejercicio 4.34: Modifique el bucle del controlador para el evaluador de modo que los pares y listas perezosos se impriman de alguna manera razonable. (¿Qué va a hacer con las listas infinitas?) También puede que necesite modificar la representación de los pares perezosos para que el evaluador pueda identificarlos para imprimirlos.

Variaciones sobre un Esquema — Computación No Determinista

En esta sección, extendemos el evaluador de Scheme para admitir un

paradigma de programación llamado *computación no determinista* incorporando en el evaluador una facilidad para admitir la búsqueda automática. Este es un cambio mucho más profundo en el lenguaje que la introducción de la evaluación perezosa en la Sección 4.2. La computación no determinista, como el procesamiento de flujos, es útil para aplicaciones de "generar y probar". Considere la tarea de comenzar con dos listas de enteros positivos y encontrar un par de enteros—uno de la primera lista y uno de la segunda lista—cuya suma sea prima. Vimos cómo manejar esto con operaciones de secuencia finita en Section 2.2.3 y con flujos infinitos en Section 3.5.3. Nuestro enfoque fue generar la secuencia de todos los posibles pares y filtrar estos para seleccionar los pares cuya suma es prima. Si realmente generamos primero toda la secuencia de pares como en el Capítulo 2, o entrelazamos la generación y el filtrado como en el Capítulo 3, es irrelevante para la imagen esencial de cómo está organizada la computación. El enfoque no determinista evoca una imagen diferente. Imagine simplemente que elegimos (de alguna manera) un número de la primera lista y un número de la segunda lista y requerimos (usando algún mecanismo) que su suma sea prima. Esto se expresa mediante el siguiente procedimiento:

(define (prime-sum-pair list1 list2) (let ((a (an-element-of list1)) (b (an-element-of list2))) (require (prime? (+ a b))) (list a b)))

Podría parecer que este procedimiento simplemente replantea el problema, en lugar de especificar una forma de resolverlo. Sin embargo, este es un programa no determinista legítimo.[42] La idea clave aquí es que las expresiones en un lenguaje no determinista pueden tener más de un valor posible. Por ejemplo, `an/element/of` podría devolver cualquier elemento de la lista dada. Nuestro evaluador de programas no determinista funcionará eligiendo automáticamente un valor posible y haciendo un seguimiento de la elección. Si un requisito posterior no se cumple, el evaluador probará una elección diferente, y seguirá probando nuevas elecciones hasta que la evaluación tenga éxito, o hasta que nos quedemos sin opciones. Al igual que el evaluador perezoso liberó al programador de los detalles de cómo se retrasan y se fuerzan los valores, el evaluador de programas no determinista liberará al programador de los detalles de cómo se hacen las elecciones. Es instructivo contrastar las diferentes imágenes del tiempo evocadas por la evaluación no determinista y el procesamiento de flujos. El procesamiento de flujos utiliza la evaluación perezosa para desacoplar el tiempo en que se ensambla el flujo de posibles respuestas del tiempo en que se producen los elementos reales del flujo. El evaluador admite la ilusión de que todas las respuestas posibles están dispuestas ante nosotros en una secuencia atemporal. Con la evaluación no determinista, una expresión representa la exploración de un conjunto de mundos posibles, cada uno determinado por un conjunto de elecciones. Algunos de los mundos posibles conducen a

callejones sin salida, mientras que otros tienen valores útiles. El evaluador de programas no determinista admite la ilusión de que el tiempo se ramifica, y que nuestros programas tienen diferentes historias de ejecución posibles. Cuando llegamos a un callejón sin salida, podemos volver a visitar un punto de elección anterior y continuar por una rama diferente. El evaluador de programas no determinista implementado a continuación se llama el evaluador `amb` porque se basa en una nueva forma especial llamada `amb`. Podemos escribir la definición anterior de `prime/sum/pair` en el bucle del controlador del evaluador `amb` (junto con las definiciones de `prime?`, `an/element/of`, y `require`) y ejecutar el procedimiento de la siguiente manera:

;;; Amb-Eval input: (prime-sum-pair '(1 3 5 8) '(20 35 110)) *;;; Starting a new problem ;;; Amb-Eval value: (3 20)*

El valor devuelto se obtuvo después de que el evaluador eligiera repetidamente elementos de cada una de las listas, hasta que se hizo una elección exitosa. El Section 4.3.1 introduce `amb` y explica cómo admite el no determinismo a través del mecanismo de búsqueda automática del evaluador. El Section 4.3.2 presenta ejemplos de programas no deterministas, y el Section 4.3.3 da los detalles de cómo implementar el evaluador `amb` modificando el evaluador ordinario de Scheme.

Amb y Búsqueda

Para extender Scheme para admitir el no determinismo, introducimos una nueva forma especial llamada `amb`.[43] La expresión

(amb $\langle$ *e* $\color{SchemeDark}{\hbox{\ttfamily\scriptsize 1}}\rangle$ $\langle$ *e* $\color{SchemeDark}{\hbox{\ttfamily\scriptsize 2}}\rangle$ … $\langle$ *e* $\color{SchemeDark}_{\hbox{\ttfamily\itshape\scriptsize n}}\rangle$)

devuelve el valor de una de las *n* expresiones ⟨*ei*⟩ "ambiguamente". Por ejemplo, la expresión

(list (amb 1 2 3) (amb 'a 'b))

puede tener seis valores posibles:

```
(1 a)  (1 b)  (2 a)  (2 b)  (3 a)  (3 b)
```

`amb` con una sola opción produce un valor ordinario (único). `amb` sin opciones —la expresión `(amb)`—es una expresión sin valores aceptables. Operacionalmente, podemos pensar en `(amb)` como una expresión que, al

evaluarse, hace que el cálculo "falle": el cálculo se aborta y no se produce ningún valor. Usando esta idea, podemos expresar el requisito de que una expresión de predicado particular `p` debe ser verdadera como sigue:

```
(define (require p) (if (not p) (amb)))
```

Con `amb` y `require`, podemos implementar el procedimiento `an/element/of` utilizado anteriormente:

```
(define (an-element-of items) (require (not (null? items))) (amb (car items) (an-element-of (cdr items))))
```

`an/element/of` falla si la lista está vacía. De lo contrario, devuelve ambiguamente ya sea el primer elemento de la lista o un elemento elegido del resto de la lista. También podemos expresar rangos infinitos de opciones. El siguiente procedimiento potencialmente devuelve cualquier entero mayor o igual que un *n* dado:

```
(define (an-integer-starting-from n) (amb n (an-integer-starting-from (+ n 1))))
```

Esto es como el procedimiento de flujo `integers/starting/from` descrito en Section 3.5.2, pero con una diferencia importante: el procedimiento de flujo devuelve un objeto que representa la secuencia de todos los enteros que comienzan con *n*, mientras que el procedimiento `amb` devuelve un solo entero.[44] Abstractamente, podemos imaginar que evaluar una expresión `amb` hace que el tiempo se divida en ramas, donde el cálculo continúa en cada rama con uno de los posibles valores de la expresión. Decimos que `amb` representa un *punto de elección no determinista*. Si tuviéramos una máquina con un número suficiente de procesadores que pudieran asignarse dinámicamente, podríamos implementar la búsqueda de una manera sencilla. La ejecución procedería como en una máquina secuencial, hasta que se encuentre una expresión `amb`. En este punto, se asignarían más procesadores y se inicializarían para continuar con todas las ejecuciones paralelas implicadas por la elección. Cada procesador procedería secuencialmente como si fuera la única elección, hasta que termine encontrando un fallo, o se subdivida aún más, o finalice.[45] Por otro lado, si tenemos una máquina que solo puede ejecutar un proceso (o unos pocos procesos concurrentes), debemos considerar las alternativas secuencialmente. Se podría imaginar modificar un evaluador para elegir al azar una rama a seguir cada vez que se encuentra con un punto de elección. La elección aleatoria, sin embargo, puede llevar fácilmente a valores fallidos. Podríamos intentar ejecutar el evaluador una y otra vez, haciendo elecciones aleatorias y esperando encontrar un valor no fallido, pero es mejor *buscar sistemáticamente* todas las

rutas de ejecución posibles. El evaluador `amb` que desarrollaremos y con el que trabajaremos en esta sección implementa una búsqueda sistemática de la siguiente manera: cuando el evaluador encuentra una aplicación de `amb`, selecciona inicialmente la primera alternativa. Esta selección puede a su vez llevar a una elección posterior. El evaluador siempre elegirá inicialmente la primera alternativa en cada punto de elección. Si una elección resulta en un fallo, entonces el evaluador retrocede *automágicamente*[46] *retrocede* al punto de elección más reciente y prueba la siguiente alternativa. Si se queda sin alternativas en cualquier punto de elección, el evaluador retrocederá al punto de elección anterior y reanudará desde allí. Este proceso conduce a una estrategia de búsqueda conocida como *búsqueda en profundidad* o *retroceso cronológico*.[47]

Bucle del controlador

El bucle del controlador para el evaluador `amb` tiene algunas propiedades inusuales. Lee una expresión e imprime el valor de la primera ejecución sin fallos, como en el ejemplo `prime/sum/pair` mostrado anteriormente. Si queremos ver el valor de la siguiente ejecución exitosa, podemos pedirle al intérprete que retroceda y intente generar una segunda ejecución sin fallos. Esto se indica escribiendo el símbolo `try/again`. Si se da cualquier expresión excepto `try/again`, el intérprete comenzará un nuevo problema, descartando las alternativas inexploradas en el problema anterior. Aquí hay una interacción de ejemplo:

;;; Entrada Amb-Eval: (prime-sum-pair '(1 3 5 8) '(20 35 110)) *;;; Empezando un nuevo problema ;;; Valor Amb-Eval: (3 20) ;;; Entrada Amb-Eval:* try-again *;;; Valor Amb-Eval: (3 110) ;;; Entrada Amb-Eval:* try-again *;;; Valor Amb-Eval: (8 35) ;;; Entrada Amb-Eval:* try-again *;;; No hay más valores de (prime-sum-pair (quote (1 3 5 8)) (quote (20 35 110))) ;;; Entrada Amb-Eval:* (prime-sum-pair '(19 27 30) '(11 36 58)) *;;; Empezando un nuevo problema ;;; Valor Amb-Eval: (30 11)*

> **Ejercicio 4.35:** Escriba un procedimiento `an/integer/between` que devuelva un entero entre dos límites dados. Esto se puede usar para implementar un procedimiento que encuentre ternas pitagóricas, es decir, ternas de enteros (i, j, k) entre los límites dados tales que $i \leq j$ y $i^2 + j^2 = k^2$, de la siguiente manera:
>
> (define (a-pythagorean-triple-between low high) (let ((i (an-integer-between low high))) (let ((j (an-integer-between i high))) (let ((k (an-integer-between j high))) (require (= (+ (* i i) (* j j)) (* k k))) (list i j k)))))

Ejercicio 4.36: El Ejercicio 3.69 discutió cómo generar el flujo de *todas* las ternas pitagóricas, sin límite superior en el tamaño de los enteros a buscar. Explique por qué simplemente reemplazar `an/integer/between` por `an/integer/starting/from` en el procedimiento del Ejercicio 3.35 no es una forma adecuada de generar ternas pitagóricas arbitrarias. Escriba un procedimiento que realmente logre esto. (Es decir, escriba un procedimiento para el cual escribir repetidamente `try/again` en principio generaría eventualmente todas las ternas pitagóricas.)

Ejercicio 4.37: Ben Bitdiddle afirma que el siguiente método para generar ternas pitagóricas es más eficiente que el del Ejercicio 4.35. ¿Tiene razón? (Pista: Considere el número de posibilidades que deben explorarse.)

(define (a-pythagorean-triple-between low high) (let ((i (an-integer-between low high)) (hsq (* high high))) (let ((j (an-integer-between i high))) (let ((ksq (+ (* i i) (* j j)))) (require (>= hsq ksq)) (let ((k (sqrt ksq))) (require (integer? k)) (list i j k))))))

Ejemplos de Programas No Deterministas

El Section 4.3.3 describe la implementación del evaluador `amb`. Primero, sin embargo, damos algunos ejemplos de cómo se puede usar. La ventaja de la programación no determinista es que podemos suprimir los detalles de cómo se lleva a cabo la búsqueda, expresando así nuestros programas en un nivel más alto de abstracción.

Acertijos Lógicos

El siguiente acertijo (tomado de Dinesman 1968) es típico de una gran clase de acertijos lógicos simples:

> Baker, Cooper, Fletcher, Miller y Smith viven en diferentes pisos de un edificio de apartamentos que solo contiene cinco pisos. Baker no vive en el piso de arriba. Cooper no vive en el piso de abajo. Fletcher no vive ni en el piso de arriba ni en el de abajo. Miller vive en un piso más alto que Cooper. Smith no vive en un piso adyacente al de Fletcher. Fletcher no vive en un piso adyacente al de Cooper. ¿Dónde vive cada uno?

Podemos determinar quién vive en cada piso de una manera sencilla enumerando todas las posibilidades e imponiendo las restricciones dadas:[48]

(define (multiple-dwelling) (let ((baker (amb 1 2 3 4 5)) (cooper (amb 1 2 3

4 5)) (fletcher (amb 1 2 3 4 5)) (miller (amb 1 2 3 4 5)) (smith (amb 1 2 3 4 5))) (require (distinct? (list baker cooper fletcher miller smith))) (require (not (= baker 5))) (require (not (= cooper 1))) (require (not (= fletcher 5))) (require (not (= fletcher 1))) (require (> miller cooper)) (require (not (= (abs (- smith fletcher)) 1))) (require (not (= (abs (- fletcher cooper)) 1))) (list (list 'baker baker) (list 'cooper cooper) (list 'fletcher fletcher) (list 'miller miller) (list 'smith smith))))

Evaluar la expresión `(multiple/dwelling)` produce el resultado

((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))

Aunque este simple procedimiento funciona, es muy lento. Los Ejercicio 4.39 y Ejercicio 4.40 discuten algunas posibles mejoras.

Ejercicio 4.38: Modifique el procedimiento de la vivienda múltiple para omitir el requisito de que Smith y Fletcher no vivan en pisos adyacentes. ¿Cuántas soluciones hay para este acertijo modificado?

Ejercicio 4.39: ¿Afecta el orden de las restricciones en el procedimiento de vivienda múltiple a la respuesta? ¿Afecta al tiempo para encontrar una respuesta? Si cree que importa, demuestre un programa más rápido obtenido del dado reordenando las restricciones. Si cree que no importa, argumente su caso.

Ejercicio 4.40: En el problema de la vivienda múltiple, ¿cuántos conjuntos de asignaciones de personas a pisos hay, tanto antes como después del requisito de que las asignaciones de pisos sean distintas? Es muy ineficiente generar todas las asignaciones posibles de personas a pisos y luego dejar que el retroceso las elimine. Por ejemplo, la mayoría de las restricciones dependen de solo una o dos de las variables persona-piso, y por lo tanto pueden imponerse antes de que se hayan seleccionado los pisos para todas las personas. Escriba y demuestre un procedimiento no determinista mucho más eficiente que resuelva este problema basándose en generar solo aquellas posibilidades que no están ya descartadas por restricciones anteriores. (Pista: Esto requerirá un nido de expresiones `let`.)

Ejercicio 4.41: Escriba un programa ordinario de Scheme para resolver el acertijo de la vivienda múltiple.

Ejercicio 4.42: Resuelva el siguiente acertijo de "Mentirosos" (de Phillips 1934): Cinco colegialas se presentaron a un examen. Sus padres, según ellas, mostraron un grado de interés indebido en el resultado. Por lo tanto, acordaron que, al escribir a casa sobre el examen, cada niña haría

una declaración verdadera y una falsa. Los siguientes son los pasajes relevantes de sus cartas:

- Betty: "Kitty fue segunda en el examen. Yo solo fui tercera".
- Ethel: "Te alegrará saber que fui la primera. Juana fue la segunda".
- Juana: "Fui tercera, y la pobre Ethel fue la última".
- Kitty: "Salí segunda. María solo fue cuarta".
- María: "Fui cuarta. El primer lugar fue para Betty".

¿Cuál fue, de hecho, el orden en que se clasificaron las cinco niñas?

Ejercicio 4.43: Use el evaluador `amb` para resolver el siguiente acertijo:[49] El padre de Mary Ann Moore tiene un yate y también lo tiene cada uno de sus cuatro amigos: el Coronel Downing, el Sr. Hall, Sir Barnacle Hood y el Dr. Parker. Cada uno de los cinco también tiene una hija y cada uno ha nombrado su yate con el nombre de una hija de uno de los otros. El yate de Sir Barnacle es el Gabrielle, el Sr. Moore es dueño del Lorna; el Sr. Hall del Rosalind. El Melissa, propiedad del Coronel Downing, lleva el nombre de la hija de Sir Barnacle. El padre de Gabrielle es dueño del yate que lleva el nombre de la hija del Dr. Parker. ¿Quién es el padre de Lorna? Intente escribir el programa para que se ejecute de manera eficiente (ver Ejercicio 4.40). Determine también cuántas soluciones hay si no se nos dice que el apellido de Mary Ann es Moore.

Ejercicio 4.44: El Ejercicio 2.42 describió el "problema de las ocho reinas" de colocar reinas en un tablero de ajedrez de modo que no se ataquen entre sí. Escriba un programa no determinista para resolver este acertijo.

Análisis del lenguaje natural

Los programas diseñados para aceptar lenguaje natural como entrada generalmente comienzan intentando *analizar* la entrada, es decir, hacer coincidir la entrada con alguna estructura gramatical. Por ejemplo, podríamos intentar reconocer oraciones simples que consisten en un artículo seguido de un sustantivo seguido de un verbo, como "El gato come". Para lograr tal análisis, debemos ser capaces de identificar las partes de la oración de las palabras individuales. Podríamos comenzar con algunas listas que clasifican varias palabras:[50]

(define nouns '(noun student professor cat class)) (define verbs '(verb studies

lectures eats sleeps)) (define articles '(article the a))

También necesitamos una *gramática*, es decir, un conjunto de reglas que describen cómo los elementos gramaticales se componen de elementos más simples. Una gramática muy simple podría estipular que una oración siempre consta de dos partes—una frase nominal seguida de un verbo—y que una frase nominal consta de un artículo seguido de un sustantivo. Con esta gramática, la oración "El gato come" se analiza de la siguiente manera:

(sentence (noun-phrase (article the) (noun cat)) (verb eats))

Podemos generar tal análisis con un programa simple que tiene procedimientos separados para cada una de las reglas gramaticales. Para analizar una oración, identificamos sus dos partes constituyentes y devolvemos una lista de estos dos elementos, etiquetada con el símbolo `sentence`:

(define (parse-sentence) (list 'sentence (parse-noun-phrase) (parse-word verbs)))

Una frase nominal, de manera similar, se analiza encontrando un artículo seguido de un sustantivo:

(define (parse-noun-phrase) (list 'noun-phrase (parse-word articles) (parse-word nouns)))

En el nivel más bajo, el análisis se reduce a comprobar repetidamente que la siguiente palabra no analizada es un miembro de la lista de palabras para la parte de la oración requerida. Para implementar esto, mantenemos una variable global `unparsed`, que es la entrada que aún no ha sido analizada. Cada vez que comprobamos una palabra, requerimos que `unparsed` no esté vacía y que comience con una palabra de la lista designada. Si es así, eliminamos esa palabra de `unparsed` y devolvemos la palabra junto con su parte de la oración (que se encuentra al principio de la lista):[51]

(define (parse-word word-list) (require (not (null? unparsed))) (require (memq (car unparsed) (cdr word-list))) (let ((found-word (car unparsed))) (set! unparsed (cdr unparsed)) (list (car word-list) found-word)))

Para iniciar el análisis, todo lo que necesitamos hacer es establecer `unparsed` para que sea toda la entrada, intentar analizar una oración y verificar que no quede nada:

(define unparsed '()) (define (parse input) (set! unparsed input) (let ((sent (parse-sentence))) (require (null? unparsed)) sent))

Ahora podemos probar el analizador y verificar que funciona para nuestra oración de prueba simple:

;;; Entrada Amb-Eval: (parse '(the cat eats)) *;;; Empezando un nuevo problema ;;; Valor Amb-Eval:*

(sentence (noun-phrase (article the) (noun cat)) (verb eats))

El evaluador `amb` es útil aquí porque es conveniente expresar las restricciones de análisis con la ayuda de `require`. Sin embargo, la búsqueda automática y el retroceso realmente valen la pena cuando consideramos gramáticas más complejas donde hay opciones sobre cómo se pueden descomponer las unidades. Agreguemos a nuestra gramática una lista de preposiciones:

(define prepositions '(prep for to in by with))

y defina una frase preposicional (p. ej., "para el gato") como una preposición seguida de una frase nominal:

(define (parse-prepositional-phrase) (list 'prep-phrase (parse-word prepositions) (parse-noun-phrase)))

Ahora podemos definir una oración como una frase nominal seguida de una frase verbal, donde una frase verbal puede ser un verbo o una frase verbal extendida por una frase preposicional:[52]

(define (parse-sentence) (list 'sentence (parse-noun-phrase) (parse-verb-phrase))) (define (parse-verb-phrase) (define (maybe-extend verb-phrase) (amb verb-phrase (maybe-extend (list 'verb-phrase verb-phrase (parse-prepositional-phrase))))) (maybe-extend (parse-word verbs)))

Mientras estamos en ello, también podemos elaborar la definición de frases nominales para permitir cosas como "un gato en la clase". Lo que solíamos llamar una frase nominal, ahora lo llamaremos una frase nominal simple, y una frase nominal ahora será ya sea una frase nominal simple o una frase nominal extendida por una frase preposicional:

(define (parse-simple-noun-phrase) (list 'simple-noun-phrase (parse-word articles) (parse-word nouns))) (define (parse-noun-phrase) (define (maybe-extend noun-phrase) (amb noun-phrase (maybe-extend (list 'noun-phrase noun-phrase (parse-prepositional-phrase))))) (maybe-extend (parse-simple-noun-phrase)))

Nuestra nueva gramática nos permite analizar oraciones más complejas. Por ejemplo

(parse '(the student with the cat sleeps in the class))

produce

(sentence (noun-phrase (simple-noun-phrase (article the) (noun student)) (prep-phrase (prep with) (simple-noun-phrase (article the) (noun cat)))) (verb-phrase (verb sleeps) (prep-phrase (prep in) (simple-noun-phrase (article the) (noun class)))))

Observe que una entrada dada puede tener más de un análisis legal. En la oración "El profesor da clases al estudiante con el gato", puede ser que el profesor esté dando clases con el gato, o que el estudiante tenga el gato. Nuestro programa no determinista encuentra ambas posibilidades:

(parse '(the professor lectures to the student with the cat))

produce

(sentence (simple-noun-phrase (article the) (noun professor)) (verb-phrase (verb-phrase (verb lectures) (prep-phrase (prep to) (simple-noun-phrase (article the) (noun student)))) (prep-phrase (prep with) (simple-noun-phrase (article the) (noun cat)))))

Pedirle al evaluador que lo intente de nuevo produce

(sentence (simple-noun-phrase (article the) (noun professor)) (verb-phrase (verb lectures) (prep-phrase (prep to) (noun-phrase (simple-noun-phrase (article the) (noun student)) (prep-phrase (prep with) (simple-noun-phrase (article the) (noun cat)))))))

Ejercicio 4.45:

((lambda (n) ((lambda (fact) (fact fact n)) (lambda (ft k) (if (= k 1) 1 (* k (ft ft (- k 1))))))) 10)

1. Verifica (evaluando la expresión) que esto realmente calcula factoriales. Diseña una expresión análoga para calcular números de Fibonacci.

2. Considere el siguiente procedimiento, que incluye definiciones internas mutuamente recursivas:

   (define (f x) (define (even? n) (if (= n 0) true (odd? (- n 1)))) (define (odd? n) (if (= n 0) false (even? (- n 1)))) (even? x))

   Completa las expresiones que faltan para completar una definición

alternativa de `f`, que no utiliza ni definiciones internas ni `letrec`:

(define (f x) ((lambda (even? odd?) (even? even? odd? x)) (lambda (ev? od? n) (if (= n 0) true (od? <??> <??> <??>))) (lambda (ev? od? n) (if (= n 0) false (ev? <??> <??> <??>)))))

Separando el Análisis Sintáctico de la Ejecución

El evaluador implementado anteriormente es simple, pero es muy ineficiente, porque el análisis sintáctico de las expresiones se entrelaza con su ejecución. Por lo tanto, si un programa se ejecuta muchas veces, su sintaxis se analiza muchas veces. Considere, por ejemplo, evaluar `(factorial 4)` usando la siguiente definición de `factorial`:

(define (factorial n) (if (= n 1) 1 (* (factorial (- n 1)) n)))

Cada vez que se llama a `factorial`, el evaluador debe determinar que el cuerpo es una expresión `if` y extraer el predicado. Solo entonces puede evaluar el predicado y despachar según su valor. Cada vez que evalúa la expresión `(* (factorial (- n 1)) n)`, o las subexpresiones `(factorial (- n 1))` y `(- n 1)`, el evaluador debe realizar el análisis de casos en `eval` para determinar que la expresión es una aplicación, y debe extraer su operador y operandos. Este análisis es costoso. Realizarlo repetidamente es un desperdicio.

Podemos transformar el evaluador para que sea significativamente más eficiente organizando las cosas de modo que el análisis sintáctico se realice solo una vez.[53] Dividimos `eval`, que toma una expresión y un entorno, en dos partes. El procedimiento `analyze` toma solo la expresión. Realiza el análisis sintáctico y devuelve un nuevo procedimiento, el *procedimiento de ejecución*, que encapsula el trabajo a realizar en la ejecución de la expresión analizada. El procedimiento de ejecución toma un entorno como argumento y completa la evaluación. Esto ahorra trabajo porque `analyze` se llamará solo una vez en una expresión, mientras que el procedimiento de ejecución puede llamarse muchas veces.

Con la separación en análisis y ejecución, `eval` ahora se convierte en

(define (eval exp env) ((analyze exp) env))

El resultado de llamar a `analyze` es el procedimiento de ejecución que se aplicará al entorno. El procedimiento `analyze` es el mismo análisis de casos que el realizado por el `eval` original de Section 4.1.1, excepto que los procedimientos a los que despachamos realizan solo análisis, no la evaluación

completa:

(define (analyze exp) (cond ((self-evaluating? exp) (analyze-self-evaluating exp)) ((quoted? exp) (analyze-quoted exp)) ((variable? exp) (analyze-variable exp)) ((assignment? exp) (analyze-assignment exp)) ((definition? exp) (analyze-definition exp)) ((if? exp) (analyze-if exp)) ((lambda? exp) (analyze-lambda exp)) ((begin? exp) (analyze-sequence (begin-actions exp))) ((cond? exp) (analyze (cond->if exp))) ((application? exp) (analyze-application exp)) (else (error "Unknown expression type: ANALYZE" exp))))

Aquí está el procedimiento de análisis sintáctico más simple, que maneja las expresiones que se autoevalúan. Devuelve un procedimiento de ejecución que ignora su argumento de entorno y simplemente devuelve la expresión:

(define (analyze-self-evaluating exp) (lambda (env) exp))

Para una expresión entrecomillada, podemos ganar un poco de eficiencia extrayendo el texto de la cita solo una vez, en la fase de análisis, en lugar de en la fase de ejecución.

(define (analyze-quoted exp) (let ((qval (text-of-quotation exp))) (lambda (env) qval)))

La búsqueda del valor de una variable todavía debe hacerse en la fase de ejecución, ya que esto depende de conocer el entorno.[54]

(define (analyze-variable exp) (lambda (env) (lookup-variable-value exp env)))

`analyze/assignment` también debe posponer el establecimiento real de la variable hasta la ejecución, cuando se haya proporcionado el entorno. Sin embargo, el hecho de que la expresión `assignment/value` pueda analizarse (recursivamente) durante el análisis es una ganancia importante en eficiencia, porque la expresión `assignment/value` ahora se analizará solo una vez. Lo mismo ocurre con las definiciones.

(define (analyze-assignment exp) (let ((var (assignment-variable exp)) (vproc (analyze (assignment-value exp)))) (lambda (env) (set-variable-value! var (vproc env) env) 'ok))) (define (analyze-definition exp) (let ((var (definition-variable exp)) (vproc (analyze (definition-value exp)))) (lambda (env) (define-variable! var (vproc env) env) 'ok)))

Para las expresiones `if`, extraemos y analizamos el predicado, el consecuente y la alternativa en el momento del análisis.

(define (analyze-if exp) (let ((pproc (analyze (if-predicate exp))) (cproc (analyze (if-consequent exp))) (aproc (analyze (if-alternative exp)))) (lambda (env) (if (true? (pproc env)) (cproc env) (aproc env)))))

Analizar una expresión `lambda` también logra una ganancia importante en eficiencia: analizamos el cuerpo de `lambda` solo una vez, aunque los procedimientos resultantes de la evaluación de la `lambda` pueden aplicarse muchas veces.

(define (analyze-lambda exp) (let ((vars (lambda-parameters exp)) (bproc (analyze-sequence (lambda-body exp)))) (lambda (env) (make-procedure vars bproc env))))

El análisis de una secuencia de expresiones (como en un `begin` o el cuerpo de una expresión `lambda`) es más complicado.[55] Cada expresión en la secuencia se analiza, produciendo un procedimiento de ejecución. Estos procedimientos de ejecución se combinan para producir un procedimiento de ejecución que toma un entorno como argumento y llama secuencialmente a cada procedimiento de ejecución individual con el entorno como argumento.

(define (analyze-sequence exps) (define (sequentially proc1 proc2) (lambda (env) (proc1 env) (proc2 env))) (define (loop first-proc rest-procs) (if (null? rest-procs) first-proc (loop (sequentially first-proc (car rest-procs)) (cdr rest-procs)))) (let ((procs (map analyze exps))) (if (null? procs) (error "Empty sequence: ANALYZE")) (loop (car procs) (cdr procs))))

Para analizar una aplicación, analizamos el operador y los operandos y construimos un procedimiento de ejecución que llama al procedimiento de ejecución del operador (para obtener el procedimiento real que se aplicará) y a los procedimientos de ejecución de los operandos (para obtener los argumentos reales). Luego los pasamos a `execute/application`, que es el análogo de `apply` en Section 4.1.1. `execute/application` difiere de `apply` en que el cuerpo del procedimiento para un procedimiento compuesto ya ha sido analizado, por lo que no hay necesidad de hacer más análisis. En su lugar, simplemente llamamos al procedimiento de ejecución para el cuerpo en el entorno extendido.

(define (analyze-application exp) (let ((fproc (analyze (operator exp))) (aprocs (map analyze (operands exp)))) (lambda (env) (execute-application (fproc env) (map (lambda (aproc) (aproc env)) aprocs))))) (define (execute-application proc args) (cond ((primitive-procedure? proc) (apply-primitive-procedure proc args)) ((compound-procedure? proc) ((procedure-body proc) (extend-environment (procedure-parameters proc) args (procedure-environment proc)))) (else (error "Unknown procedure type: EXECUTE-

APPLICATION" proc))))

Nuestro nuevo evaluador utiliza las mismas estructuras de datos, procedimientos de sintaxis y procedimientos de soporte en tiempo de ejecución que en las secciones Sección 4.1.2, Sección 4.1.3 y Sección 4.1.4.

Ejercicio 4.22: Extienda el evaluador de esta sección para que admita la forma especial `let`. (Ver Ejercicio 4.6.)

Ejercicio 4.23: Alyssa P. Hacker no entiende por qué `analyze/sequence` necesita ser tan complicado. Todos los demás procedimientos de análisis son transformaciones directas de los procedimientos de evaluación correspondientes (o cláusulas `eval`) en Section 4.1.1. Ella esperaba que `analyze/sequence` se viera así:

(define (analyze-sequence exps) (define (execute-sequence procs env) (cond ((null? (cdr procs)) ((car procs) env)) (else ((car procs) env) (execute-sequence (cdr procs) env)))) (let ((procs (map analyze exps))) (if (null? procs) (error "Empty sequence: ANALYZE")) (lambda (env) (execute-sequence procs env))))

Eva Lu Ator le explica a Alyssa que la versión del texto hace más trabajo de evaluación de una secuencia en el momento del análisis. El procedimiento de ejecución de secuencias de Alyssa, en lugar de tener las llamadas a los procedimientos de ejecución individuales incorporadas, recorre los procedimientos para llamarlos: en efecto, aunque las expresiones individuales de la secuencia han sido analizadas, la secuencia en sí no lo ha sido.

Compare las dos versiones de `analyze/sequence`. Por ejemplo, considere el caso común (típico de los cuerpos de los procedimientos) donde la secuencia tiene una sola expresión. ¿Qué trabajo hará el procedimiento de ejecución producido por el programa de Alyssa? ¿Y el procedimiento de ejecución producido por el programa del texto anterior? ¿Cómo se comparan las dos versiones para una secuencia con dos expresiones?

Ejercicio 4.24: Diseñe y lleve a cabo algunos experimentos para comparar la velocidad del evaluador metacircular original con la versión de esta sección. Use sus resultados para estimar la fracción de tiempo que se dedica al análisis frente a la ejecución para varios procedimientos.

Variaciones sobre un Esquema — Evaluación

Perezosa

Ahora que tenemos un evaluador expresado como un programa Lisp, podemos experimentar con opciones alternativas en el diseño de lenguajes simplemente modificando el evaluador. De hecho, a menudo se inventan nuevos lenguajes escribiendo primero un evaluador que incrusta el nuevo lenguaje dentro de un lenguaje de alto nivel existente. Por ejemplo, si deseamos discutir algún aspecto de una modificación propuesta a Lisp con otro miembro de la comunidad Lisp, podemos proporcionar un evaluador que encarne el cambio. El destinatario puede entonces experimentar con el nuevo evaluador y enviar comentarios como modificaciones adicionales. No solo la base de implementación de alto nivel facilita la prueba y depuración del evaluador; además, la incrustación permite al diseñador tomar prestadas[56] características del lenguaje subyacente, al igual que nuestro evaluador de Lisp incrustado usa primitivas y estructuras de control del Lisp subyacente. Solo más tarde (si es que alguna vez) el diseñador necesita tomarse la molestia de construir una implementación completa en un lenguaje de bajo nivel o en hardware. En esta sección y la siguiente exploraremos algunas variaciones de Scheme que proporcionan un poder expresivo adicional significativo.

Orden Normal y Orden Aplicativo

En Section 1.1, donde comenzamos nuestra discusión sobre los modelos de evaluación, notamos que Scheme es un lenguaje de *orden aplicativo*, es decir, que todos los argumentos de los procedimientos de Scheme se evalúan cuando se aplica el procedimiento. En contraste, los lenguajes de *orden normal* retrasan la evaluación de los argumentos de los procedimientos hasta que se necesitan los valores reales de los argumentos. Retrasar la evaluación de los argumentos de los procedimientos hasta el último momento posible (por ejemplo, hasta que sean requeridos por una operación primitiva) se llama *evaluación perezosa*.[57] Considere el procedimiento

```
(define (try a b) (if (= a 0) 1 b))
```

Evaluar `(try 0 (/ 1 0))` genera un error en Scheme. Con la evaluación perezosa, no habría ningún error. Evaluar la expresión devolvería 1, porque el argumento `(/ 1 0)` nunca se evaluaría.

Un ejemplo que explota la evaluación perezosa es la definición de un procedimiento `unless`

```
(define (unless condition usual-value exceptional-value) (if condition
```

exceptional-value usual-value))

que se puede usar en expresiones como

(unless (= b 0) (/ a b) (begin (display "exception: returning 0") 0))

Esto no funcionará en un lenguaje de orden aplicativo porque tanto el valor usual como el valor excepcional se evaluarán antes de que se llame a `unless` (compare el Ejercicio 1.6). Una ventaja de la evaluación perezosa es que algunos procedimientos, como `unless`, pueden realizar cálculos útiles incluso si la evaluación de algunos de sus argumentos produciría errores o no terminaría.

Si el cuerpo de un procedimiento se introduce antes de que un argumento haya sido evaluado, decimos que el procedimiento es *no estricto* en ese argumento. Si el argumento se evalúa antes de que se introduzca el cuerpo del procedimiento, decimos que el procedimiento es *estricto* en ese argumento.[58] En un lenguaje puramente de orden aplicativo, todos los procedimientos son estrictos en cada argumento. En un lenguaje puramente de orden normal, todos los procedimientos compuestos son no estrictos en cada argumento, y los procedimientos primitivos pueden ser estrictos o no estrictos. También hay lenguajes (ver Ejercicio 4.31) que dan a los programadores un control detallado sobre la estrictez de los procedimientos que definen.

Un ejemplo sorprendente de un procedimiento que puede hacerse útilmente no estricto es `cons` (o, en general, casi cualquier constructor para estructuras de datos). Se pueden realizar cálculos útiles, combinando elementos para formar estructuras de datos y operando sobre las estructuras de datos resultantes, incluso si los valores de los elementos no se conocen. Tiene perfecto sentido, por ejemplo, calcular la longitud de una lista sin conocer los valores de los elementos individuales de la lista. Explotaremos esta idea en Section 4.2.3 para implementar los flujos del Capítulo 3 como listas formadas por pares `cons` no estrictos.

> **Ejercicio 4.25:** Supongamos que (en Scheme de orden aplicativo ordinario) definimos `unless` como se muestra arriba y luego definimos `factorial` en términos de `unless` como
>
> (define (factorial n) (unless (= n 1) (* n (factorial (- n 1))) 1))
>
> ¿Qué sucede si intentamos evaluar `(factorial 5)`? ¿Funcionarán nuestras definiciones en un lenguaje de orden normal?

Ejercicio 4.26: Ben Bitdiddle y Alyssa P. Hacker no están de acuerdo

sobre la importancia de la evaluación perezosa para implementar cosas como `unless`. Ben señala que es posible implementar `unless` en orden aplicativo como una forma especial. Alyssa replica que, si se hiciera eso, `unless` sería simplemente sintaxis, no un procedimiento que podría usarse junto con procedimientos de orden superior. Complete los detalles de ambos lados del argumento. Muestre cómo implementar `unless` como una expresión derivada (como `cond` o `let`), y dé un ejemplo de una situación en la que podría ser útil tener `unless` disponible como un procedimiento, en lugar de como una forma especial.

Un Intérprete con Evaluación Perezosa

En esta sección implementaremos un lenguaje de orden normal que es igual a Scheme excepto que los procedimientos compuestos no son estrictos en cada argumento. Los procedimientos primitivos seguirán siendo estrictos. No es difícil modificar el evaluador de Section 4.1.1 para que el lenguaje que interpreta se comporte de esta manera. Casi todos los cambios requeridos se centran en la aplicación de procedimientos.

La idea básica es que, al aplicar un procedimiento, el intérprete debe determinar qué argumentos deben evaluarse y cuáles deben retrasarse. Los argumentos retrasados no se evalúan; en su lugar, se transforman en objetos llamados *thunks*.[59] El thunk debe contener la información requerida para producir el valor del argumento cuando sea necesario, como si hubiera sido evaluado en el momento de la aplicación. Por lo tanto, el thunk debe contener la expresión del argumento y el entorno en el que se está evaluando la aplicación del procedimiento.

El proceso de evaluar la expresión en un thunk se llama *forzar*.[60] En general, un thunk se forzará solo cuando se necesite su valor: cuando se pase a un procedimiento primitivo que usará el valor del thunk; cuando sea el valor de un predicado de un condicional; y cuando sea el valor de un operador que está a punto de aplicarse como un procedimiento. Una opción de diseño que tenemos disponible es si *memoizar* o no los thunks, como hicimos con los objetos retardados en Section 3.5.1. Con la memoización, la primera vez que se fuerza un thunk, almacena el valor que se calcula. Las forzadas posteriores simplemente devuelven el valor almacenado sin repetir el cálculo. Haremos que nuestro intérprete memoize, porque esto es más eficiente para muchas aplicaciones. Aquí, sin embargo, hay consideraciones delicadas.[61]

Modificando el evaluador

La principal diferencia entre el evaluador perezoso y el de Section 4.1 está en

el manejo de las aplicaciones de procedimientos en `eval` y `apply`.

La cláusula `application?` de `eval` se convierte en

((application? exp) (apply (actual-value (operator exp) env) (operands exp) env))

Esto es casi lo mismo que la cláusula `application?` de `eval` en Section 4.1.1. Sin embargo, para la evaluación perezosa, llamamos a `apply` con las expresiones de los operandos, en lugar de los argumentos producidos al evaluarlos. Como necesitaremos el entorno para construir thunks si los argumentos deben retrasarse, también debemos pasarlo. Todavía evaluamos el operador, porque `apply` necesita el procedimiento real que se aplicará para despachar según su tipo (primitivo versus compuesto) y aplicarlo.

Siempre que necesitemos el valor real de una expresión, usamos

(define (actual-value exp env) (force-it (eval exp env)))

en lugar de solo `eval`, para que si el valor de la expresión es un thunk, se fuerce.

Nuestra nueva versión de `apply` también es casi la misma que la versión de la Sección 4.1.1. La diferencia es que `eval` ha pasado expresiones de operandos no evaluadas: para los procedimientos primitivos (que son estrictos), evaluamos todos los argumentos antes de aplicar la primitiva; para los procedimientos compuestos (que no son estrictos), retrasamos todos los argumentos antes de aplicar el procedimiento.

(define (apply procedure arguments env) (cond ((primitive-procedure? procedure) (apply-primitive-procedure procedure (list-of-arg-values arguments env))) ; cambiado ((compound-procedure? procedure) (eval-sequence (procedure-body procedure) (extend-environment (procedure-parameters procedure) (list-of-delayed-args arguments env) ; cambiado (procedure-environment procedure)))) (else (error "Unknown procedure type: APPLY" procedure))))

Los procedimientos que procesan los argumentos son como `list/of/values` de Section 4.1.1, excepto que `list/of/delayed/args` retrasa los argumentos en lugar de evaluarlos, y `list/of/arg/values` usa `actual/value` en lugar de `eval`:

(define (list-of-arg-values exps env) (if (no-operands? exps) '() (cons (actual-value (first-operand exps) env) (list-of-arg-values (rest-operands exps) env)))) (define (list-of-delayed-args exps env) (if (no-operands? exps) '() (cons

(delay-it (first-operand exps) env) (list-of-delayed-args (rest-operands exps) env))))

El otro lugar donde debemos cambiar el evaluador es en el manejo de `if`, donde debemos usar `actual/value` en lugar de `eval` para obtener el valor de la expresión del predicado antes de probar si es verdadero o falso:

(define (eval-if exp env) (if (true? (actual-value (if-predicate exp) env)) (eval (if-consequent exp) env) (eval (if-alternative exp) env)))

Finalmente, debemos cambiar el procedimiento `driver/loop` (Section 4.1.4) para usar `actual/value` en lugar de `eval`, de modo que si un valor retardado se propaga de vuelta al bucle de lectura-evaluación-impresión, se forzará antes de ser impreso. También cambiamos los avisos para indicar que este es el evaluador perezoso:

(define input-prompt ";;; L-Eval input:") (define output-prompt ";;; L-Eval value:") (define (driver-loop) (prompt-for-input input-prompt) (let ((input (read))) (let ((output (actual-value input the-global-environment))) (announce-output output-prompt) (user-print output))) (driver-loop))

Con estos cambios hechos, podemos iniciar el evaluador y probarlo. La evaluación exitosa de la expresión `try` discutida en Section 4.2.1 indica que el intérprete está realizando una evaluación perezosa:

(define the-global-environment (setup-environment)) (driver-loop) *;;; L-Eval input:* (define (try a b) (if (= a 0) 1 b)) *;;; L-Eval value: ok ;;; L-Eval input:* (try 0 (/ 1 0)) *;;; L-Eval value: 1*

Representando thunks

Nuestro evaluador debe encargarse de crear thunks cuando se aplican procedimientos a argumentos y de forzar estos thunks más tarde. Un thunk debe empaquetar una expresión junto con el entorno, para que el argumento pueda producirse más tarde. Para forzar el thunk, simplemente extraemos la expresión y el entorno del thunk y evaluamos la expresión en el entorno. Usamos `actual/value` en lugar de `eval` para que, en caso de que el valor de la expresión sea a su vez un thunk, forcemos ese, y así sucesivamente, hasta que lleguemos a algo que no sea un thunk:

(define (force-it obj) (if (thunk? obj) (actual-value (thunk-exp obj) (thunk-env obj)) obj))

Una forma fácil de empaquetar una expresión con un entorno es hacer una lista que contenga la expresión y el entorno. Por lo tanto, creamos un thunk

de la siguiente manera:

(define (delay-it exp env) (list 'thunk exp env)) (define (thunk? obj) (tagged-list? obj 'thunk)) (define (thunk-exp thunk) (cadr thunk)) (define (thunk-env thunk) (caddr thunk))

En realidad, lo que queremos para nuestro intérprete no es exactamente esto, sino más bien thunks que han sido memoizados. Cuando se fuerza un thunk, lo convertiremos en un thunk evaluado reemplazando la expresión almacenada con su valor y cambiando la etiqueta `thunk` para que pueda reconocerse como ya evaluado.[62]

(define (evaluated-thunk? obj) (tagged-list? obj 'evaluated-thunk)) (define (thunk-value evaluated-thunk) (cadr evaluated-thunk)) (define (force-it obj) (cond ((thunk? obj) (let ((result (actual-value (thunk-exp obj) (thunk-env obj)))) (set-car! obj 'evaluated-thunk) (set-car! (cdr obj) result) ; reemplazar `exp` con su valor (set-cdr! (cdr obj) '()) ; olvidar el `env` innecesario result)) ((evaluated-thunk? obj) (thunk-value obj)) (else obj)))

Nótese que el mismo procedimiento `delay/it` funciona tanto con como sin memoización.

Ejercicio 4.27: Supongamos que escribimos las siguientes definiciones en el evaluador perezoso:

(define count 0) (define (id x) (set! count (+ count 1)) x)

Dé los valores que faltan en la siguiente secuencia de interacciones y explique sus respuestas.[63]

(define w (id (id 10))) *;;; L-Eval input:* count *;;; L-Eval value:* *< respuesta >* *;;; L-Eval input:* w *;;; L-Eval value:* *< respuesta >* *;;; L-Eval input:* count *;;; L-Eval value:* *< respuesta >*

Ejercicio 4.28: `eval` usa `actual/value` en lugar de `eval` para evaluar el operador antes de pasarlo a `apply`, con el fin de forzar el valor del operador. Dé un ejemplo que demuestre la necesidad de este forzamiento.

Ejercicio 4.29: Muestre un programa que esperaría que se ejecutara mucho más lentamente sin memoización que con memoización. Además, considere la siguiente interacción, donde el procedimiento `id` se define como en el Ejercicio 4.27 y `count` comienza en 0:

(define (square x) (* x x)) *;;; L-Eval input:* (square (id 10)) *;;; L-Eval value:* *< respuesta >* *;;; L-Eval input:* count *;;; L-Eval value:*

〈 *respuesta* 〉

Dé las respuestas tanto cuando el evaluador memoiza como cuando no lo hace.

Ejercicio 4.30: Cy D. Fect, un programador de C reformado, está preocupado de que algunos efectos secundarios nunca ocurran, porque el evaluador perezoso no fuerza las expresiones en una secuencia. Dado que el valor de una expresión en una secuencia que no sea la última no se usa (la expresión está ahí solo por su efecto, como asignar a una variable o imprimir), no puede haber un uso posterior de este valor (por ejemplo, como argumento de un procedimiento primitivo) que haga que se fuerce. Cy piensa, por lo tanto, que al evaluar secuencias, debemos forzar todas las expresiones de la secuencia excepto la final. Propone modificar `eval/sequence` de Section 4.1.1 para usar `actual/value` en lugar de `eval`:

(define (eval-sequence exps env) (cond ((last-exp? exps) (eval (first-exp exps) env)) (else (actual-value (first-exp exps) env) (eval-sequence (rest-exps exps) env))))

1. Ben Bitdiddle cree que Cy está equivocado. Le muestra a Cy el procedimiento `for/each` descrito en el Ejercicio 2.23, que da un ejemplo importante de una secuencia con efectos secundarios:

   (define (for-each proc items) (if (null? items) 'done (begin (proc (car items)) (for-each proc (cdr items)))))

   Afirma que el evaluador del texto (con el `eval/sequence` original) maneja esto correctamente:

   ;;; L-Eval input: (for-each (lambda (x) (newline) (display x)) (list 57 321 88)) *57 321 88 ;;; L-Eval value: done*

   Explique por qué Ben tiene razón sobre el comportamiento de `for/each`.

2. Cy está de acuerdo en que Ben tiene razón sobre el ejemplo de `for/each`, pero dice que ese no es el tipo de programa en el que estaba pensando cuando propuso su cambio a `eval/sequence`. Define los siguientes dos procedimientos en el evaluador perezoso:

   (define (p1 x) (set! x (cons x '(2))) x) (define (p2 x) (define (p e) e x) (p (set! x (cons x '(2)))))

   ¿Cuáles son los valores de `(p1 1)` y `(p2 1)` con el `eval/sequence`

original? ¿Cuáles serían los valores con el cambio propuesto por Cy a `eval/sequence`?

3. Cy también señala que cambiar `eval/sequence` como él propone no afecta el comportamiento del ejemplo de la parte a. Explique por qué esto es cierto.

4. ¿Cómo cree que deberían tratarse las secuencias en el evaluador perezoso? ¿Le gusta el enfoque de Cy, el enfoque del texto o algún otro enfoque?

Ejercicio 4.31: El enfoque adoptado en esta sección es algo desagradable, porque introduce un cambio incompatible con Scheme. Podría ser más agradable implementar la evaluación perezosa como una extensión *compatible hacia arriba*, es decir, para que los programas ordinarios de Scheme funcionen como antes. Podemos hacer esto extendiendo la sintaxis de las declaraciones de procedimientos para permitir que el usuario controle si los argumentos deben ser retrasados o no. Ya que estamos en ello, también podríamos darle al usuario la opción de elegir entre retrasar con y sin memoización. Por ejemplo, la definición

(define (f a (b lazy) c (d lazy-memo)) …)

definiría `f` como un procedimiento de cuatro argumentos, donde el primer y tercer argumento se evalúan cuando se llama al procedimiento, el segundo argumento se retrasa, y el cuarto argumento se retrasa y se memoiza. Así, las definiciones de procedimientos ordinarios producirán el mismo comportamiento que el Scheme ordinario, mientras que agregar la declaración `lazy/memo` a cada parámetro de cada procedimiento compuesto producirá el comportamiento del evaluador perezoso definido en esta sección. Diseñe e implemente los cambios requeridos para producir tal extensión a Scheme. Tendrá que implementar nuevos procedimientos de sintaxis para manejar la nueva sintaxis para `define`. También debe hacer arreglos para que `eval` o `apply` determinen cuándo los argumentos deben ser retrasados, y para forzar o retrasar los argumentos en consecuencia, y debe hacer arreglos para que el forzamiento memoice o no, según corresponda.

Flujos como Listas Perezosas

En Section 3.5.1, mostramos cómo implementar flujos como listas retardadas. Introdujimos las formas especiales `delay` y `cons/stream`, que nos permitieron construir una "promesa" de calcular el `cdr` de un flujo, sin cumplir realmente esa promesa hasta más tarde. Podríamos usar esta técnica

general de introducir formas especiales cada vez que necesitemos más control sobre el proceso de evaluación, pero esto es incómodo. Por un lado, una forma especial no es un objeto de primera clase como un procedimiento, por lo que no podemos usarla junto con procedimientos de orden superior.[64] Además, nos vimos obligados a crear flujos como un nuevo tipo de objeto de datos similar pero no idéntico a las listas, y esto nos obligó a reimplementar muchas operaciones de lista ordinarias (`map`, `append`, y así sucesivamente) para su uso con flujos.

Con la evaluación perezosa, los flujos y las listas pueden ser idénticos, por lo que no hay necesidad de formas especiales o de operaciones separadas para listas y flujos. Todo lo que necesitamos hacer es arreglar las cosas para que `cons` no sea estricto. Una forma de lograr esto es extender el evaluador perezoso para permitir primitivas no estrictas, e implementar `cons` como una de estas. Una forma más fácil es recordar (Section 2.1.3) que no hay una necesidad fundamental de implementar `cons` como una primitiva en absoluto. En su lugar, podemos representar los pares como procedimientos:[65]

(define (cons x y) (lambda (m) (m x y))) (define (car z) (z (lambda (p q) p))) (define (cdr z) (z (lambda (p q) q)))

En términos de estas operaciones básicas, las definiciones estándar de las operaciones de lista funcionarán tanto con listas infinitas (flujos) como con las finitas, y las operaciones de flujo pueden implementarse como operaciones de lista. Aquí hay algunos ejemplos:

(define (list-ref items n) (if (= n 0) (car items) (list-ref (cdr items) (- n 1)))) (define (map proc items) (if (null? items) '() (cons (proc (car items)) (map proc (cdr items))))) (define (scale-list items factor) (map (lambda (x) (* x factor)) items)) (define (add-lists list1 list2) (cond ((null? list1) list2) ((null? list2) list1) (else (cons (+ (car list1) (car list2)) (add-lists (cdr list1) (cdr list2)))))) (define ones (cons 1 ones)) (define integers (cons 1 (add-lists ones integers))) *;;; L-Eval input:* (list-ref integers 17) *;;; L-Eval value: 18*

Nótese que estas listas perezosas son aún más perezosas que los flujos de Capítulo 3: El `car` de la lista, así como el `cdr`, se retrasa.[66] De hecho, incluso acceder al `car` o al `cdr` de un par perezoso no necesita forzar el valor de un elemento de la lista. El valor se forzará solo cuando sea realmente necesario, por ejemplo, para usarlo como argumento de una primitiva, o para imprimirlo como respuesta.

Los pares perezosos también ayudan con el problema que surgió con los flujos en Section 3.5.4, donde descubrimos que la formulación de modelos de flujo de sistemas con bucles puede requerir que salpiquemos nuestros

programas con operaciones explícitas de `delay`, más allá de las proporcionadas por `cons/stream`. Con la evaluación perezosa, todos los argumentos de los procedimientos se retrasan uniformemente. Por ejemplo, podemos implementar procedimientos para integrar listas y resolver ecuaciones diferenciales como pretendíamos originalmente en Section 3.5.4:

(define (integral integrand initial-value dt) (define int (cons initial-value (add-lists (scale-list integrand dt) int))) int) (define (solve f y0 dt) (define y (integral dy y0 dt)) (define dy (map f y)) y) *;;; L-Eval input:* (list-ref (solve (lambda (x) x) 1 0.001) 1000) *;;; L-Eval value: 2.716924*

Ejercicio 4.32: Dé algunos ejemplos que ilustren la diferencia entre los flujos del Capítulo 3 y las listas perezosas "más perezosas" descritas en esta sección. ¿Cómo puede aprovechar esta pereza adicional?

Ejercicio 4.33: Ben Bitdiddle prueba la implementación de la lista perezosa dada anteriormente evaluando la expresión:

(car ’(a b c))

Para su sorpresa, esto produce un error. Después de pensarlo un poco, se da cuenta de que las "listas" obtenidas al leer expresiones entrecomilladas son diferentes de las listas manipuladas por las nuevas definiciones de `cons`, `car` y `cdr`. Modifique el tratamiento que el evaluador da a las expresiones entrecomilladas para que las listas entrecomilladas escritas en el bucle del controlador produzcan listas perezosas verdaderas.

Ejercicio 4.34: Modifique el bucle del controlador para el evaluador de modo que los pares y listas perezosos se impriman de alguna manera razonable. (¿Qué va a hacer con las listas infinitas?) También puede que necesite modificar la representación de los pares perezosos para que el evaluador pueda identificarlos para imprimirlos.

Variaciones sobre un Esquema — Computación No Determinista

En esta sección, extendemos el evaluador de Scheme para admitir un paradigma de programación llamado *computación no determinista* incorporando en el evaluador una facilidad para admitir la búsqueda automática. Este es un cambio mucho más profundo en el lenguaje que la introducción de la evaluación perezosa en la Sección 4.2.

La computación no determinista, como el procesamiento de flujos, es útil

para aplicaciones de "generar y probar". Considere la tarea de comenzar con dos listas de enteros positivos y encontrar un par de enteros—uno de la primera lista y uno de la segunda lista—cuya suma sea prima. Vimos cómo manejar esto con operaciones de secuencia finita en Section 2.2.3 y con flujos infinitos en Section 3.5.3. Nuestro enfoque fue generar la secuencia de todos los posibles pares y filtrar estos para seleccionar los pares cuya suma es prima. Si realmente generamos primero toda la secuencia de pares como en el Capítulo 2, o entrelazamos la generación y el filtrado como en el Capítulo 3, es irrelevante para la imagen esencial de cómo está organizada la computación.

El enfoque no determinista evoca una imagen diferente. Imagine simplemente que elegimos (de alguna manera) un número de la primera lista y un número de la segunda lista y requerimos (usando algún mecanismo) que su suma sea prima. Esto se expresa mediante el siguiente procedimiento:

(define (prime-sum-pair list1 list2) (let ((a (an-element-of list1)) (b (an-element-of list2))) (require (prime? (+ a b))) (list a b)))

Podría parecer que este procedimiento simplemente replantea el problema, en lugar de especificar una forma de resolverlo. Sin embargo, este es un programa no determinista legítimo.[67]

La idea clave aquí es que las expresiones en un lenguaje no determinista pueden tener más de un valor posible. Por ejemplo, `an/element/of` podría devolver cualquier elemento de la lista dada. Nuestro evaluador de programas no determinista funcionará eligiendo automáticamente un valor posible y haciendo un seguimiento de la elección. Si un requisito posterior no se cumple, el evaluador probará una elección diferente, y seguirá probando nuevas elecciones hasta que la evaluación tenga éxito, o hasta que nos quedemos sin opciones. Al igual que el evaluador perezoso liberó al programador de los detalles de cómo se retrasan y se fuerzan los valores, el evaluador de programas no determinista liberará al programador de los detalles de cómo se hacen las elecciones.

Es instructivo contrastar las diferentes imágenes del tiempo evocadas por la evaluación no determinista y el procesamiento de flujos. El procesamiento de flujos utiliza la evaluación perezosa para desacoplar el tiempo en que se ensambla el flujo de posibles respuestas del tiempo en que se producen los elementos reales del flujo. El evaluador admite la ilusión de que todas las respuestas posibles están dispuestas ante nosotros en una secuencia atemporal. Con la evaluación no determinista, una expresión representa la exploración de un conjunto de mundos posibles, cada uno determinado por un conjunto de elecciones. Algunos de los mundos posibles conducen a

callejones sin salida, mientras que otros tienen valores útiles. El evaluador de programas no determinista admite la ilusión de que el tiempo se ramifica, y que nuestros programas tienen diferentes historias de ejecución posibles. Cuando llegamos a un callejón sin salida, podemos volver a visitar un punto de elección anterior y continuar por una rama diferente.

El evaluador de programas no determinista implementado a continuación se llama el evaluador `amb` porque se basa en una nueva forma especial llamada `amb`. Podemos escribir la definición anterior de `prime/sum/pair` en el bucle del controlador del evaluador `amb` (junto con las definiciones de `prime?`, `an/element/of`, y `require`) y ejecutar el procedimiento de la siguiente manera:

;;; Entrada Amb-Eval: (prime-sum-pair '(1 3 5 8) '(20 35 110)) *;;; Empezando un nuevo problema ;;; Valor Amb-Eval: (3 20)*

El valor devuelto se obtuvo después de que el evaluador eligiera repetidamente elementos de cada una de las listas, hasta que se hizo una elección exitosa.

El Section 4.3.1 introduce `amb` y explica cómo admite el no determinismo a través del mecanismo de búsqueda automática del evaluador. El Section 4.3.2 presenta ejemplos de programas no deterministas, y el Section 4.3.3 da los detalles de cómo implementar el evaluador `amb` modificando el evaluador ordinario de Scheme.

Amb y Búsqueda

Para extender Scheme para admitir el no determinismo, introducimos una nueva forma especial llamada `amb`.[68] La expresión

(amb < *e* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$ < *e* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$ … < *e* $\color{SchemeDark}_{\hbox{\ttfamily\itshape\scriptsize n}}\rangle$)

devuelve el valor de una de las *n* expresiones ⟨*ei*⟩ "ambiguamente". Por ejemplo, la expresión

(list (amb 1 2 3) (amb 'a 'b))

puede tener seis valores posibles:

```
(1 a)  (1 b)  (2 a)  (2 b)  (3 a)  (3 b)
```

`amb` con una sola opción produce un valor ordinario (único).

`amb` sin opciones—la expresión `(amb)`—es una expresión sin valores aceptables. Operacionalmente, podemos pensar en `(amb)` como una expresión que, al evaluarse, hace que el cálculo "falle": el cálculo se aborta y no se produce ningún valor. Usando esta idea, podemos expresar el requisito de que una expresión de predicado particular `p` debe ser verdadera como sigue:

```
(define (require p) (if (not p) (amb)))
```

Con `amb` y `require`, podemos implementar el procedimiento `an/element/of` utilizado anteriormente:

```
(define (an-element-of items) (require (not (null? items))) (amb (car items) (an-element-of (cdr items))))
```

`an/element/of` falla si la lista está vacía. De lo contrario, devuelve ambiguamente ya sea el primer elemento de la lista o un elemento elegido del resto de la lista.

También podemos expresar rangos infinitos de opciones. El siguiente procedimiento potencialmente devuelve cualquier entero mayor o igual que un *n* dado:

```
(define (an-integer-starting-from n) (amb n (an-integer-starting-from (+ n 1))))
```

Esto es como el procedimiento de flujo `integers/starting/from` descrito en Section 3.5.2, pero con una diferencia importante: el procedimiento de flujo devuelve un objeto que representa la secuencia de todos los enteros que comienzan con *n*, mientras que el procedimiento `amb` devuelve un solo entero.[69]

Abstractamente, podemos imaginar que evaluar una expresión `amb` hace que el tiempo se divida en ramas, donde el cálculo continúa en cada rama con uno de los posibles valores de la expresión. Decimos que `amb` representa un *punto de elección no determinista*. Si tuviéramos una máquina con un número suficiente de procesadores que pudieran asignarse dinámicamente, podríamos implementar la búsqueda de una manera sencilla. La ejecución procedería como en una máquina secuencial, hasta que se encuentre una expresión `amb`. En este punto, se asignarían más procesadores y se inicializarían para continuar con todas las ejecuciones paralelas implicadas por la elección. Cada procesador procedería secuencialmente como si fuera la única elección, hasta que termine encontrando un fallo, o se subdivida aún más, o finalice.[70]

Por otro lado, si tenemos una máquina que solo puede ejecutar un proceso (o

unos pocos procesos concurrentes), debemos considerar las alternativas secuencialmente. Se podría imaginar modificar un evaluador para elegir al azar una rama a seguir cada vez que se encuentra con un punto de elección. La elección aleatoria, sin embargo, puede llevar fácilmente a valores fallidos. Podríamos intentar ejecutar el evaluador una y otra vez, haciendo elecciones aleatorias y esperando encontrar un valor no fallido, pero es mejor *buscar sistemáticamente* todas las rutas de ejecución posibles. El evaluador `amb` que desarrollaremos y con el que trabajaremos en esta sección implementa una búsqueda sistemática de la siguiente manera: cuando el evaluador encuentra una aplicación de `amb`, selecciona inicialmente la primera alternativa. Esta selección puede a su vez llevar a una elección posterior. El evaluador siempre elegirá inicialmente la primera alternativa en cada punto de elección. Si una elección resulta en un fallo, entonces el evaluador retrocede *automágicamente*[71] *retrocede* al punto de elección más reciente y prueba la siguiente alternativa. Si se queda sin alternativas en cualquier punto de elección, el evaluador retrocederá al punto de elección anterior y reanudará desde allí. Este proceso conduce a una estrategia de búsqueda conocida como *búsqueda en profundidad* o *retroceso cronológico*.[72]

Bucle del controlador

El bucle del controlador para el evaluador `amb` tiene algunas propiedades inusuales. Lee una expresión e imprime el valor de la primera ejecución sin fallos, como en el ejemplo `prime/sum/pair` mostrado anteriormente. Si queremos ver el valor de la siguiente ejecución exitosa, podemos pedirle al intérprete que retroceda y intente generar una segunda ejecución sin fallos. Esto se indica escribiendo el símbolo `try/again`. Si se da cualquier expresión excepto `try/again`, el intérprete comenzará un nuevo problema, descartando las alternativas inexploradas en el problema anterior. Aquí hay una interacción de ejemplo:

;;; Entrada Amb-Eval: (prime-sum-pair '(1 3 5 8) '(20 35 110)) *;;; Empezando un nuevo problema ;;; Valor Amb-Eval: (3 20)*

;;; Entrada Amb-Eval: try-again *;;; Valor Amb-Eval: (3 110)*

;;; Entrada Amb-Eval: try-again *;;; Valor Amb-Eval: (8 35)*

;;; Entrada Amb-Eval: try-again *;;; No hay más valores de (prime-sum-pair (quote (1 3 5 8)) (quote (20 35 110)))*

;;; Entrada Amb-Eval: (prime-sum-pair '(19 27 30) '(11 36 58)) *;;; Empezando un nuevo problema ;;; Valor Amb-Eval: (30 11)*

Ejercicio 4.35: Escriba un procedimiento `an/integer/between` que devuelva un entero entre dos límites dados. Esto se puede usar para implementar un procedimiento que encuentre ternas pitagóricas, es decir, ternas de enteros (i, j, k) entre los límites dados tales que $i \leq j$ y $i^2 + j^2 = k^2$, de la siguiente manera:

(define (a-pythagorean-triple-between low high) (let ((i (an-integer-between low high))) (let ((j (an-integer-between i high))) (let ((k (an-integer-between j high))) (require (= (+ (* i i) (* j j)) (* k k))) (list i j k)))))

Ejercicio 4.36: El Ejercicio 3.69 discutió cómo generar el flujo de *todas* las ternas pitagóricas, sin límite superior en el tamaño de los enteros a buscar. Explique por qué simplemente reemplazar `an/integer/between` por `an/integer/starting/from` en el procedimiento del Ejercicio 3.35 no es una forma adecuada de generar ternas pitagóricas arbitrarias. Escriba un procedimiento que realmente logre esto. (Es decir, escriba un procedimiento para el cual escribir repetidamente `try/again` en principio generaría eventualmente todas las ternas pitagóricas.)

Ejercicio 4.37: Ben Bitdiddle afirma que el siguiente método para generar ternas pitagóricas es más eficiente que el del Ejercicio 4.35. ¿Tiene razón? (Pista: Considere el número de posibilidades que deben explorarse.)

(define (a-pythagorean-triple-between low high) (let ((i (an-integer-between low high)) (hsq (* high high))) (let ((j (an-integer-between i high))) (let ((ksq (+ (* i i) (* j j)))) (require (>= hsq ksq)) (let ((k (sqrt ksq))) (require (integer? k)) (list i j k))))))

Ejemplos de Programas No Deterministas

El Section 4.3.3 describe la implementación del evaluador `amb`. Primero, sin embargo, damos algunos ejemplos de cómo se puede usar. La ventaja de la programación no determinista es que podemos suprimir los detalles de cómo se lleva a cabo la búsqueda, expresando así nuestros programas en un nivel más alto de abstracción.

Acertijos Lógicos

El siguiente acertijo (tomado de Dinesman 1968) es típico de una gran clase de acertijos lógicos simples:

Baker, Cooper, Fletcher, Miller y Smith viven en diferentes pisos de un edificio de apartamentos que solo contiene cinco pisos. Baker no vive en el piso de arriba. Cooper no vive en el piso de abajo. Fletcher no vive ni en el piso de arriba ni en el de abajo. Miller vive en un piso más alto que Cooper. Smith no vive en un piso adyacente al de Fletcher. Fletcher no vive en un piso adyacente al de Cooper. ¿Dónde vive cada uno?

Podemos determinar quién vive en cada piso de una manera sencilla enumerando todas las posibilidades e imponiendo las restricciones dadas:[73]

```
(define (multiple-dwelling)
  (let ((baker (amb 1 2 3 4 5))
        (cooper (amb 1 2 3 4 5))
        (fletcher (amb 1 2 3 4 5))
        (miller (amb 1 2 3 4 5))
        (smith (amb 1 2 3 4 5)))
    (require (distinct? (list baker cooper fletcher miller smith)))
    (require (not (= baker 5)))
    (require (not (= cooper 1)))
    (require (not (= fletcher 5)))
    (require (not (= fletcher 1)))
    (require (> miller cooper))
    (require (not (= (abs (- smith fletcher)) 1)))
    (require (not (= (abs (- fletcher cooper)) 1)))
    (list (list 'baker baker)
          (list 'cooper cooper)
          (list 'fletcher fletcher)
          (list 'miller miller)
          (list 'smith smith))))
```

Evaluar la expresión `(multiple/dwelling)` produce el resultado

```
((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))
```

Aunque este simple procedimiento funciona, es muy lento. Los Ejercicio 4.39 y Ejercicio 4.40 discuten algunas posibles mejoras.

Ejercicio 4.38: Modifique el procedimiento de la vivienda múltiple para omitir el requisito de que Smith y Fletcher no vivan en pisos adyacentes. ¿Cuántas soluciones hay para este acertijo modificado?

Ejercicio 4.39: ¿Afecta el orden de las restricciones en el procedimiento de vivienda múltiple a la respuesta? ¿Afecta al tiempo para encontrar una respuesta? Si cree que importa, demuestre un programa más rápido obtenido del dado reordenando las restricciones. Si cree que no importa, argumente su caso.

Ejercicio 4.40: En el problema de la vivienda múltiple, ¿cuántos conjuntos de asignaciones de personas a pisos hay, tanto antes como después del requisito de que las asignaciones de pisos sean distintas? Es muy ineficiente generar todas las asignaciones posibles de personas a pisos y luego dejar que el retroceso las elimine. Por ejemplo, la mayoría de las restricciones dependen de solo una o dos de las variables persona-piso, y por lo tanto pueden imponerse antes de que se hayan seleccionado los pisos para todas las personas. Escriba y demuestre un procedimiento no determinista mucho más eficiente que resuelva este problema

basándose en generar solo aquellas posibilidades que no están ya descartadas por restricciones anteriores. (Pista: Esto requerirá un nido de expresiones `let`.)

Ejercicio 4.41: Escriba un programa ordinario de Scheme para resolver el acertijo de la vivienda múltiple.

Ejercicio 4.42: Resuelva el siguiente acertijo de "Mentirosos" (de Phillips 1934):

Cinco colegialas se presentaron a un examen. Sus padres, según ellas, mostraron un grado de interés indebido en el resultado. Por lo tanto, acordaron que, al escribir a casa sobre el examen, cada niña haría una declaración verdadera y una falsa. Los siguientes son los pasajes relevantes de sus cartas:

- Betty: "Kitty fue segunda en el examen. Yo solo fui tercera".
- Ethel: "Te alegrará saber que fui la primera. Juana fue la segunda".
- Juana: "Fui tercera, y la pobre Ethel fue la última".
- Kitty: "Salí segunda. María solo fue cuarta".
- María: "Fui cuarta. El primer lugar fue para Betty".

¿Cuál fue, de hecho, el orden en que se clasificaron las cinco niñas?

Ejercicio 4.43: Use el evaluador `amb` para resolver el siguiente acertijo:[74]

El padre de Mary Ann Moore tiene un yate y también lo tiene cada uno de sus cuatro amigos: el Coronel Downing, el Sr. Hall, Sir Barnacle Hood y el Dr. Parker. Cada uno de los cinco también tiene una hija y cada uno ha nombrado su yate con el nombre de una hija de uno de los otros. El yate de Sir Barnacle es el Gabrielle, el Sr. Moore es dueño del Lorna; el Sr. Hall del Rosalind. El Melissa, propiedad del Coronel Downing, lleva el nombre de la hija de Sir Barnacle. El padre de Gabrielle es dueño del yate que lleva el nombre de la hija del Dr. Parker. ¿Quién es el padre de Lorna?

Intente escribir el programa para que se ejecute de manera eficiente (ver Ejercicio 4.40). Determine también cuántas soluciones hay si no se nos dice que el apellido de Mary Ann es Moore.

Ejercicio 4.44: El Ejercicio 2.42 describió el "problema de las ocho

reinas" de colocar reinas en un tablero de ajedrez de modo que no se ataquen entre sí. Escriba un programa no determinista para resolver este acertijo.

Análisis del lenguaje natural

Los programas diseñados para aceptar lenguaje natural como entrada generalmente comienzan intentando *analizar* la entrada, es decir, hacer coincidir la entrada con alguna estructura gramatical. Por ejemplo, podríamos intentar reconocer oraciones simples que consisten en un artículo seguido de un sustantivo seguido de un verbo, como "El gato come". Para lograr tal análisis, debemos ser capaces de identificar las partes de la oración de las palabras individuales. Podríamos comenzar con algunas listas que clasifican varias palabras:[75]

(define nouns '(noun student professor cat class)) (define verbs '(verb studies lectures eats sleeps)) (define articles '(article the a))

También necesitamos una *gramática*, es decir, un conjunto de reglas que describen cómo los elementos gramaticales se componen de elementos más simples. Una gramática muy simple podría estipular que una oración siempre consta de dos partes—una frase nominal seguida de un verbo—y que una frase nominal consta de un artículo seguido de un sustantivo. Con esta gramática, la oración "El gato come" se analiza de la siguiente manera:

(sentence (noun-phrase (article the) (noun cat)) (verb eats))

Podemos generar tal análisis con un programa simple que tiene procedimientos separados para cada una de las reglas gramaticales. Para analizar una oración, identificamos sus dos partes constituyentes y devolvemos una lista de estos dos elementos, etiquetada con el símbolo `sentence`:

(define (parse-sentence) (list 'sentence (parse-noun-phrase) (parse-word verbs)))

Una frase nominal, de manera similar, se analiza encontrando un artículo seguido de un sustantivo:

(define (parse-noun-phrase) (list 'noun-phrase (parse-word articles) (parse-word nouns)))

En el nivel más bajo, el análisis se reduce a comprobar repetidamente que la siguiente palabra no analizada es un miembro de la lista de palabras para la parte de la oración requerida. Para implementar esto, mantenemos una

variable global `*unparsed*`, que es la entrada que aún no ha sido analizada. Cada vez que comprobamos una palabra, requerimos que `*unparsed*` no esté vacía y que comience con una palabra de la lista designada. Si es así, eliminamos esa palabra de `*unparsed*` y devolvemos la palabra junto con su parte de la oración (que se encuentra al principio de la lista):[76]

(define (parse-word word-list) (require (not (null? *unparsed*))) (require (memq (car *unparsed*) (cdr word-list))) (let ((found-word (car *unparsed*))) (set! *unparsed* (cdr *unparsed*)) (list (car word-list) found-word)))

Para iniciar el análisis, todo lo que necesitamos hacer es establecer `*unparsed*` para que sea toda la entrada, intentar analizar una oración y verificar que no quede nada:

(define *unparsed* '()) (define (parse input) (set! *unparsed* input) (let ((sent (parse-sentence))) (require (null? *unparsed*)) sent))

Ahora podemos probar el analizador y verificar que funciona para nuestra oración de prueba simple:

;;; Entrada Amb-Eval: (parse '(the cat eats)) *;;; Empezando un nuevo problema*
;;; Valor Amb-Eval:

(sentence (noun-phrase (article the) (noun cat)) (verb eats))

El evaluador `amb` es útil aquí porque es conveniente expresar las restricciones de análisis con la ayuda de `require`. Sin embargo, la búsqueda automática y el retroceso realmente valen la pena cuando consideramos gramáticas más complejas donde hay opciones sobre cómo se pueden descomponer las unidades.

Agreguemos a nuestra gramática una lista de preposiciones:

(define prepositions '(prep for to in by with))

y defina una frase preposicional (p. ej., "para el gato") como una preposición seguida de una frase nominal:

(define (parse-prepositional-phrase) (list 'prep-phrase (parse-word prepositions) (parse-noun-phrase)))

Ahora podemos definir una oración como una frase nominal seguida de una frase verbal, donde una frase verbal puede ser un verbo o una frase verbal extendida por una frase preposicional:[77]

(define (parse-sentence) (list 'sentence (parse-noun-phrase) (parse-verb-phrase))) (define (parse-verb-phrase) (define (maybe-extend verb-phrase) (amb verb-phrase (maybe-extend (list 'verb-phrase verb-phrase (parse-prepositional-phrase))))) (maybe-extend (parse-word verbs)))

Mientras estamos en ello, también podemos elaborar la definición de frases nominales para permitir cosas como "un gato en la clase". Lo que solíamos llamar una frase nominal, ahora lo llamaremos una frase nominal simple, y una frase nominal ahora será ya sea una frase nominal simple o una frase nominal extendida por una frase preposicional:

(define (parse-simple-noun-phrase) (list 'simple-noun-phrase (parse-word articles) (parse-word nouns))) (define (parse-noun-phrase) (define (maybe-extend noun-phrase) (amb noun-phrase (maybe-extend (list 'noun-phrase noun-phrase (parse-prepositional-phrase))))) (maybe-extend (parse-simple-noun-phrase)))

Nuestra nueva gramática nos permite analizar oraciones más complejas. Por ejemplo

(parse '(the student with the cat sleeps in the class))

produce

(sentence (noun-phrase (simple-noun-phrase (article the) (noun student)) (prep-phrase (prep with) (simple-noun-phrase (article the) (noun cat)))) (verb-phrase (verb sleeps) (prep-phrase (prep in) (simple-noun-phrase (article the) (noun class)))))

Observe que una entrada dada puede tener más de un análisis legal. En la oración "El profesor da clases al estudiante con el gato", puede ser que el profesor esté dando clases con el gato, o que el estudiante tenga el gato. Nuestro programa no determinista encuentra ambas posibilidades:

(parse '(the professor lectures to the student with the cat))

produce

(sentence (simple-noun-phrase (article the) (noun professor)) (verb-phrase (verb-phrase (verb lectures) (prep-phrase (prep to) (simple-noun-phrase (article the) (noun student)))) (prep-phrase (prep with) (simple-noun-phrase (article the) (noun cat)))))

Pedirle al evaluador que lo intente de nuevo produce

(sentence (simple-noun-phrase (article the) (noun professor)) (verb-phrase (verb lectures) (prep-phrase (prep to) (noun-phrase (simple-noun-phrase (article the) (noun student)) (prep-phrase (prep with) (simple-noun-phrase (article the) (noun cat)))))))

Ejercicio 4.45: Con la gramática dada anteriormente, la siguiente oración puede analizarse de cinco maneras diferentes: "El profesor da clases al estudiante en la clase con el gato". Dé los cinco análisis y explique las diferencias de matices de significado entre ellos.

Ejercicio 4.46: Los evaluadores en Section 4.1 y Section 4.2 no determinan en qué orden se evalúan los operandos. Veremos que el evaluador `amb` los evalúa de izquierda a derecha. Explique por qué nuestro programa de análisis no funcionaría si los operandos se evaluaran en algún otro orden.

Ejercicio 4.47: Louis Reasoner sugiere que, dado que una frase verbal es un verbo o una frase verbal seguida de una frase preposicional, sería mucho más sencillo definir el procedimiento `parse/verb/phrase` de la siguiente manera (y de manera similar para las frases nominales):

(define (parse-verb-phrase) (amb (parse-word verbs) (list 'verb-phrase (parse-verb-phrase) (parse-prepositional-phrase))))

¿Funciona esto? ¿Cambia el comportamiento del programa si intercambiamos el orden de las expresiones en el `amb`?

Ejercicio 4.48: Extienda la gramática dada anteriormente para manejar oraciones más complejas. Por ejemplo, podría extender las frases nominales y verbales para incluir adjetivos y adverbios, o podría manejar oraciones compuestas. [78]

Ejercicio 4.49: Alyssa P. Hacker está más interesada en generar oraciones interesantes que en analizarlas. Ella razona que simplemente cambiando el procedimiento `parse/word` para que ignore la "oración de entrada" y en su lugar siempre tenga éxito y genere una palabra apropiada, podemos usar los programas que habíamos construido para el análisis para hacer la generación en su lugar. Implemente la idea de Alyssa y muestre la primera media docena o más de oraciones generadas.[79]

Implementando el Evaluador `amb`

La evaluación de una expresión ordinaria de Scheme puede devolver un

valor, puede no terminar nunca o puede señalar un error. En Scheme no determinista, la evaluación de una expresión puede además resultar en el descubrimiento de un callejón sin salida, en cuyo caso la evaluación debe retroceder a un punto de elección anterior. La interpretación de Scheme no determinista se complica por este caso extra.

Construiremos el evaluador `amb` para Scheme no determinista modificando el evaluador de análisis de Section 4.1.7.[80] Al igual que en el evaluador de análisis, la evaluación de una expresión se logra llamando a un procedimiento de ejecución producido por el análisis de esa expresión. La diferencia entre la interpretación de Scheme ordinario y la interpretación de Scheme no determinista estará enteramente en los procedimientos de ejecución.

Procedimientos de ejecución y continuaciones

Recuerde que los procedimientos de ejecución para el evaluador ordinario toman un argumento: el entorno de ejecución. En contraste, los procedimientos de ejecución en el evaluador `amb` toman tres argumentos: el entorno y dos procedimientos llamados *procedimientos de continuación*. La evaluación de una expresión terminará llamando a una de estas dos continuaciones: si la evaluación resulta en un valor, se llama a la *continuación de éxito* con ese valor; si la evaluación resulta en el descubrimiento de un callejón sin salida, se llama a la *continuación de fallo*. Construir y llamar a las continuaciones apropiadas es el mecanismo por el cual el evaluador no determinista implementa el retroceso.

El trabajo de la continuación de éxito es recibir un valor y continuar con el cálculo. Junto con ese valor, a la continuación de éxito se le pasa otra continuación de fallo, que se llamará posteriormente si el uso de ese valor conduce a un callejón sin salida.

El trabajo de la continuación de fallo es probar otra rama del proceso no determinista. La esencia del lenguaje no determinista está en el hecho de que las expresiones pueden representar elecciones entre alternativas. La evaluación de tal expresión debe proceder con una de las elecciones alternativas indicadas, aunque no se sepa de antemano qué elecciones conducirán a resultados aceptables. Para lidiar con esto, el evaluador elige una de las alternativas y pasa este valor a la continuación de éxito. Junto con este valor, el evaluador construye y pasa una continuación de fallo que puede llamarse más tarde para elegir una alternativa diferente.

Se desencadena un fallo durante la evaluación (es decir, se llama a una continuación de fallo) cuando un programa de usuario rechaza

explícitamente la línea de ataque actual (por ejemplo, una llamada a `require` puede resultar en la ejecución de `(amb)`, una expresión que siempre falla—ver Section 4.3.1). La continuación de fallo disponible en ese punto hará que el punto de elección más reciente elija otra alternativa. Si no hay más alternativas que considerar en ese punto de elección, se desencadena un fallo en un punto de elección anterior, y así sucesivamente. Las continuaciones de fallo también son invocadas por el bucle del controlador en respuesta a una solicitud `try/again`, para encontrar otro valor de la expresión.

Además, si una operación de efecto secundario (como la asignación a una variable) ocurre en una rama del proceso resultante de una elección, puede ser necesario, cuando el proceso encuentra un callejón sin salida, deshacer el efecto secundario antes de hacer una nueva elección. Esto se logra haciendo que la operación de efecto secundario produzca una continuación de fallo que deshaga el efecto secundario y propague el fallo.

En resumen, las continuaciones de fallo son construidas por

- expresiones `amb`—para proporcionar un mecanismo para hacer elecciones alternativas si la elección actual hecha por la expresión `amb` conduce a un callejón sin salida;
- el controlador de nivel superior—para proporcionar un mecanismo para informar del fallo cuando las elecciones se agotan;
- asignaciones—para interceptar fallos y deshacer asignaciones durante el retroceso.

Los fallos se inician solo cuando se encuentra un callejón sin salida. Esto ocurre

- si el programa de usuario ejecuta `(amb)`;
- si el usuario escribe `try/again` en el controlador de nivel superior.

Las continuaciones de fallo también se llaman durante el procesamiento de un fallo:

- Cuando la continuación de fallo creada por una asignación termina de deshacer un efecto secundario, llama a la continuación de fallo que interceptó, para propagar el fallo de vuelta al punto de elección que condujo a esta asignación o al nivel superior.
- Cuando la continuación de fallo para un `amb` se queda sin opciones, llama

a la continuación de fallo que se le dio originalmente al `amb`, para propagar el fallo de vuelta al punto de elección anterior o al nivel superior.

Estructura del evaluador

Los procedimientos de sintaxis y representación de datos para el evaluador `amb`, y también el procedimiento básico `analyze`, son idénticos a los del evaluador de Section 4.1.7, excepto por el hecho de que necesitamos procedimientos de sintaxis adicionales para reconocer la forma especial `amb`:[81]

(define (amb? exp) (tagged-list? exp 'amb)) (define (amb-choices exp) (cdr exp))

También debemos agregar a la distribución en `analyze` una cláusula que reconocerá esta forma especial y generará un procedimiento de ejecución apropiado:

((amb? exp) (analyze-amb exp))

El procedimiento de nivel superior `ambeval` (similar a la versión de `eval` dada en Section 4.1.7) analiza la expresión dada y aplica el procedimiento de ejecución resultante al entorno dado, junto con dos continuaciones dadas:

(define (ambeval exp env succeed fail) ((analyze exp) env succeed fail))

Una continuación de éxito es un procedimiento de dos argumentos: el valor recién obtenido y otra continuación de fracaso que se utilizará si ese valor conduce a un fracaso posterior. Una continuación de fracaso es un procedimiento sin argumentos. Por lo tanto, la forma general de un procedimiento de ejecución es

(lambda (env succeed fail) ;; `succeed` is `(lambda (value fail) ...)` ;; `fail` is `(lambda () ...)` ...)

Por ejemplo, ejecutar

(ambeval ⟨ *exp* ⟩ the-global-environment (lambda (value fail) value) (lambda () 'failed))

intentará evaluar la expresión dada y devolverá o bien el valor de la expresión (si la evaluación tiene éxito) o el símbolo `failed` (si la evaluación falla). La llamada a `ambeval` en el bucle del controlador que se muestra a continuación utiliza procedimientos de continuación mucho más

complicados, que continúan el bucle y admiten la solicitud `try/again`.

La mayor parte de la complejidad del evaluador `amb` resulta de la mecánica de pasar las continuaciones mientras los procedimientos de ejecución se llaman entre sí. Al revisar el siguiente código, debería comparar cada uno de los procedimientos de ejecución con el procedimiento correspondiente del evaluador ordinario dado en Section 4.1.7.

Expresiones simples

Los procedimientos de ejecución para los tipos más simples de expresiones son esencialmente los mismos que para el evaluador ordinario, excepto por la necesidad de gestionar las continuaciones. Los procedimientos de ejecución simplemente tienen éxito con el valor de la expresión, pasando la continuación de fallo que se les pasó.

(define (analyze-self-evaluating exp) (lambda (env succeed fail) (succeed exp fail))) (define (analyze-quoted exp) (let ((qval (text-of-quotation exp))) (lambda (env succeed fail) (succeed qval fail)))) (define (analyze-variable exp) (lambda (env succeed fail) (succeed (lookup-variable-value exp env) fail))) (define (analyze-lambda exp) (let ((vars (lambda-parameters exp)) (bproc (analyze-sequence (lambda-body exp)))) (lambda (env succeed fail) (succeed (make-procedure vars bproc env) fail))))

Nótese que buscar una variable siempre 'tiene éxito'. Si `lookup/variable/value` no encuentra la variable, señala un error, como de costumbre. Tal "fallo" indica un error de programa—una referencia a una variable no ligada; no es una indicación de que debamos probar otra elección no determinista en lugar de la que se está probando actualmente.

Condicionales y secuencias

Los condicionales también se manejan de manera similar a como lo hace el evaluador ordinario. El procedimiento de ejecución generado por `analyze/if` invoca el procedimiento de ejecución del predicado `pproc` con una continuación de éxito que comprueba si el valor del predicado es verdadero y pasa a ejecutar ya sea el consecuente o la alternativa. Si la ejecución de `pproc` falla, se llama a la continuación de fallo original para la expresión `if`.

(define (analyze-if exp) (let ((pproc (analyze (if-predicate exp))) (cproc (analyze (if-consequent exp))) (aproc (analyze (if-alternative exp)))) (lambda (env succeed fail) (pproc env ;; continuación de éxito para evaluar el predicado ;; para obtener `pred/value` (lambda (pred-value fail2) (if (true? pred-value) (cproc env succeed fail2) (aproc env succeed fail2))) ;;

continuación de fallo para evaluar el predicado fail))))

Las secuencias también se manejan de la misma manera que en el evaluador anterior, excepto por las maquinaciones en el subprocedimiento `sequentially` que se requieren para pasar las continuaciones. A saber, para ejecutar secuencialmente `a` y luego `b`, llamamos a `a` con una continuación de éxito que llama a `b`.

(define (analyze-sequence exps) (define (sequentially a b) (lambda (env succeed fail) (a env ;; continuación de éxito para llamar a `a` (lambda (a-value fail2) (b env succeed fail2)) ;; continuación de fallo para llamar a `a` fail))) (define (loop first-proc rest-procs) (if (null? rest-procs) first-proc (loop (sequentially first-proc (car rest-procs)) (cdr rest-procs)))) (let ((procs (map analyze exps))) (if (null? procs) (error "Empty sequence: ANALYZE")) (loop (car procs) (cdr procs))))

Definiciones y asignaciones

Las definiciones son otro caso en el que debemos esforzarnos para gestionar las continuaciones, porque es necesario evaluar la expresión del valor de la definición antes de definir realmente la nueva variable. Para lograr esto, el procedimiento de ejecución del valor de la definición `vproc` se llama con el entorno, una continuación de éxito y la continuación de fracaso. Si la ejecución de `vproc` tiene éxito, obteniendo un valor `val` para la variable definida, la variable se define y el éxito se propaga:

(define (analyze-definition exp) (let ((var (definition-variable exp)) (vproc (analyze (definition-value exp)))) (lambda (env succeed fail) (vproc env (lambda (val fail2) (define-variable! var val env) (succeed 'ok fail2)) fail))))

Las asignaciones son más interesantes. Este es el primer lugar donde realmente usamos las continuaciones, en lugar de solo pasarlas. El procedimiento de ejecución para las asignaciones comienza como el de las definiciones. Primero intenta obtener el nuevo valor que se asignará a la variable. Si esta evaluación de `vproc` falla, la asignación falla.

Sin embargo, si `vproc` tiene éxito y continuamos para hacer la asignación, debemos considerar la posibilidad de que esta rama del cálculo pueda fallar más tarde, lo que requerirá que retrocedamos de la asignación. Por lo tanto, debemos organizar el deshacer la asignación como parte del proceso de retroceso.[82]

Esto se logra dándole a `vproc` una continuación de éxito (marcada con el comentario "*1*" a continuación) que guarda el valor antiguo de la variable

antes de asignar el nuevo valor a la variable y continuar desde la asignación. La continuación de fracaso que se pasa junto con el valor de la asignación (marcada con el comentario "*2*" a continuación) restaura el valor antiguo de la variable antes de continuar con el fracaso. Es decir, una asignación exitosa proporciona una continuación de fracaso que interceptará un fracaso posterior; cualquier fracaso que de otro modo habría llamado a `fail2` llama a este procedimiento en su lugar, para deshacer la asignación antes de llamar realmente a `fail2`.

(define (analyze-assignment exp) (let ((var (assignment-variable exp)) (vproc (analyze (assignment-value exp)))) (lambda (env succeed fail) (vproc env (lambda (val fail2) ; *1* (let ((old-value (lookup-variable-value var env))) (set-variable-value! var val env) (succeed 'ok (lambda () ; *2* (set-variable-value! var old-value env) (fail2))))) fail))))

Aplicaciones de procedimiento

El procedimiento de ejecución para las aplicaciones no contiene ideas nuevas excepto por la complejidad técnica de gestionar las continuaciones. Esta complejidad surge en `analyze/application`, debido a la necesidad de hacer un seguimiento de las continuaciones de éxito y fracaso mientras evaluamos los operandos. Usamos un procedimiento `get/args` para evaluar la lista de operandos, en lugar de un simple `map` como en el evaluador ordinario.

(define (analyze-application exp) (let ((fproc (analyze (operator exp))) (aprocs (map analyze (operands exp)))) (lambda (env succeed fail) (fproc env (lambda (proc fail2) (get-args aprocs env (lambda (args fail3) (execute-application proc args succeed fail3)) fail2)) fail))))

En `get/args`, observe cómo el `cdr` de la lista de procedimientos de ejecución `aproc` y el `cons` de la lista resultante de `args` se logran llamando a cada `aproc` de la lista con una continuación de éxito que llama recursivamente a `get/args`. Cada una de estas llamadas recursivas a `get/args` tiene una continuación de éxito cuyo valor es el `cons` del argumento recién obtenido en la lista de argumentos acumulados:

(define (get-args aprocs env succeed fail) (if (null? aprocs) (succeed '() fail) ((car aprocs) env ;; continuación de éxito para este `aproc` (lambda (arg fail2) (get-args (cdr aprocs) env ;; continuación de éxito para ;; llamada recursiva a `get/args` (lambda (args fail3) (succeed (cons arg args) fail3)) fail2)) fail)))

La aplicación real del procedimiento, que es realizada por `execute/appli/cation`, se logra de la misma manera que en el evaluador ordinario, excepto

por la necesidad de gestionar las continuaciones.

(define (execute-application proc args succeed fail) (cond ((primitive-procedure? proc) (succeed (apply-primitive-procedure proc args) fail)) ((compound-procedure? proc) ((procedure-body proc) (extend-environment (procedure-parameters proc) args (procedure-environment proc)) succeed fail)) (else (error "Unknown procedure type: EXECUTE-APPLICATION" proc))))

Evaluando expresiones `amb`

La forma especial `amb` es el elemento clave en el lenguaje no determinista. Aquí vemos la esencia del proceso de interpretación y la razón de llevar un registro de las continuaciones. El procedimiento de ejecución para `amb` define un bucle `try/next` que cicla a través de los procedimientos de ejecución para todos los valores posibles de la expresión `amb`. Cada procedimiento de ejecución se llama con una continuación de fallo que probará el siguiente. Cuando no hay más alternativas que probar, toda la expresión `amb` falla.

(define (analyze-amb exp) (let ((cprocs (map analyze (amb-choices exp)))) (lambda (env succeed fail) (define (try-next choices) (if (null? choices) (fail) ((car choices) env succeed (lambda () (try-next (cdr choices)))))) (try-next cprocs))))

Bucle del controlador

El bucle del controlador para el evaluador `amb` es complejo, debido al mecanismo que permite al usuario volver a intentar la evaluación de una expresión. El controlador utiliza un procedimiento llamado `internal/loop`, que toma como argumento un procedimiento `try/again`. La intención es que llamar a `try/again` deba pasar a la siguiente alternativa no probada en la evaluación no determinista. `internal/loop` o llama a `try/again` en respuesta a que el usuario escriba `try/again` en el bucle del controlador, o bien inicia una nueva evaluación llamando a `ambeval`.

La continuación de fallo para esta llamada a `ambeval` informa al usuario de que no hay más valores y reinicia el bucle del controlador.

La continuación de éxito para la llamada a `ambeval` es más sutil. Imprimimos el valor obtenido y luego invocamos de nuevo el bucle interno con un procedimiento `try/again` que podrá probar la siguiente alternativa. Este procedimiento `next/alternative` es el segundo argumento que se pasó a la continuación de éxito. Normalmente, pensamos en este segundo argumento como una continuación de fallo que se usará si la rama de evaluación actual

falla más tarde. En este caso, sin embargo, hemos completado una evaluación exitosa, por lo que podemos invocar la rama alternativa de "fallo" para buscar evaluaciones exitosas adicionales.

(define input-prompt ";;; Entrada Amb-Eval:") (define output-prompt ";;; Valor Amb-Eval:")

(define (driver-loop) (define (internal-loop try-again) (prompt-for-input input-prompt) (let ((input (read))) (if (eq? input 'try-again) (try-again) (begin (newline) (display ";;; Empezando un nuevo problema ") (ambeval input the-global-environment ;; éxito de `ambeval` (lambda (val next-alternative) (announce-output output-prompt) (user-print val) (internal-loop next-alternative)) ;; fallo de `ambeval` (lambda () (announce-output ";;; No hay más valores de") (user-print input) (driver-loop))))))) (internal-loop (lambda () (newline) (display ";;; No hay ningún problema actual") (driver-loop))))

La llamada inicial a `internal/loop` usa un procedimiento `try/again` que se queja de que no hay ningún problema actual y reinicia el bucle del controlador. Este es el comportamiento que ocurrirá si el usuario escribe `try/again` cuando no hay ninguna evaluación en curso.

Ejercicio 4.50: Implemente una nueva forma especial `ramb` que sea como `amb` excepto que busca alternativas en un orden aleatorio, en lugar de de izquierda a derecha. Muestre cómo esto puede ayudar con el problema de Alyssa en el Ejercicio 4.49.

Ejercicio 4.51: Implemente un nuevo tipo de asignación llamada `permanent/set!` que no se deshace en caso de fallo. Por ejemplo, podemos elegir dos elementos distintos de una lista y contar el número de intentos necesarios para hacer una elección exitosa de la siguiente manera:

(define count 0) (let ((x (an-element-of '(a b c))) (y (an-element-of '(a b c)))) (permanent-set! count (+ count 1)) (require (not (eq? x y))) (list x y count)) *;;; Starting a new problem ;;; Amb-Eval value: (a b 2) ;;; Amb-Eval input:* try-again *;;; Amb-Eval value: (a c 3)*

¿Qué valores se habrían mostrado si hubiéramos usado `set!` aquí en lugar de `permanent/set!`?

Ejercicio 4.52: Implemente una nueva construcción llamada `if/fail` que permita al usuario capturar el fallo de una expresión. `if/fail` toma dos expresiones. Evalúa la primera expresión como de costumbre y devuelve el resultado como de costumbre si la evaluación tiene éxito. Si la evaluación falla, sin embargo, se devuelve el valor de la segunda

expresión, como en el siguiente ejemplo:

;;; Entrada Amb-Eval: (if-fail (let ((x (an-element-of '(1 3 5)))) (require (even? x)) x) 'all-odd) *;;; Empezando un nuevo problema ;;; Valor Amb-Eval: all-odd*

;;; Entrada Amb-Eval: (if-fail (let ((x (an-element-of '(1 3 5 8)))) (require (even? x)) x) 'all-odd) *;;; Empezando un nuevo problema ;;; Valor Amb-Eval: 8*

Ejercicio 4.53: Con `permanent/set!` como se describe en el Ejercicio 4.51 y `if/fail` como en Ejercicio 4.52, ¿cuál será el resultado de evaluar

(let ((pairs '())) (if-fail (let ((p (prime-sum-pair '(1 3 5 8) '(20 35 110)))) (permanent-set! pairs (cons p pairs)) (amb)) pairs))

Ejercicio 4.54: Si no nos hubiéramos dado cuenta de que `require` podría implementarse como un procedimiento ordinario que usa `amb`, para ser definido por el usuario como parte de un programa no determinista, hubiéramos tenido que implementarlo como una forma especial. Esto requeriría procedimientos de sintaxis

(define (require? exp) (tagged-list? exp 'require)) (define (require-predicate exp) (cadr exp))

y una nueva cláusula en el despacho en `analyze`

((require? exp) (analyze-require exp))

así como el procedimiento `analyze/require` que maneja las expresiones `require`. Complete la siguiente definición de `analyze/require`.

(define (analyze-require exp) (let ((pproc (analyze (require-predicate exp)))) (lambda (env succeed fail) (pproc env (lambda (pred-value fail2) (if < ?? > < ?? > (succeed 'ok fail2))) fail))))

Programación Lógica

En el Capítulo 1 enfatizamos que la informática trata con el conocimiento imperativo (cómo hacer), mientras que las matemáticas tratan con el conocimiento declarativo (qué es). De hecho, los lenguajes de programación requieren que el programador exprese el conocimiento de una forma que indique los métodos paso a paso para resolver problemas particulares. Por otro lado, los lenguajes de alto nivel proporcionan, como parte de la

implementación del lenguaje, una cantidad sustancial de conocimiento metodológico que libera al usuario de la preocupación por numerosos detalles de cómo progresará un cómputo especificado.

La mayoría de los lenguajes de programación, incluido Lisp, se organizan en torno al cálculo de los valores de las funciones matemáticas. Los lenguajes orientados a expresiones (como Lisp, Fortran y Algol) aprovechan el "juego de palabras" de que una expresión que describe el valor de una función también puede interpretarse como un medio para calcular ese valor. Debido a esto, la mayoría de los lenguajes de programación están fuertemente sesgados hacia los cálculos unidireccionales (cálculos con entradas y salidas bien definidas). Sin embargo, existen lenguajes de programación radicalmente diferentes que relajan este sesgo. Vimos un ejemplo de este tipo en Section 3.3.5, donde los objetos de cálculo eran restricciones aritméticas. En un sistema de restricciones, la dirección y el orden del cálculo no están tan bien especificados; al llevar a cabo un cálculo, el sistema debe por lo tanto proporcionar un conocimiento de "cómo hacerlo" más detallado del que sería el caso con un cálculo aritmético ordinario. Esto no significa, sin embargo, que el usuario quede totalmente liberado de la responsabilidad de proporcionar conocimiento imperativo. Hay muchas redes de restricciones que implementan el mismo conjunto de restricciones, y el usuario debe elegir del conjunto de redes matemáticamente equivalentes una red adecuada para especificar un cálculo particular.

El evaluador de programas no determinista de Section 4.3 también se aleja de la visión de que la programación consiste en construir algoritmos para calcular funciones unidireccionales. En un lenguaje no determinista, las expresiones pueden tener más de un valor y, como resultado, el cálculo trata con relaciones en lugar de con funciones de un solo valor. La programación lógica extiende esta idea combinando una visión relacional de la programación con un tipo poderoso de coincidencia de patrones simbólicos llamado *unificación*.[83]

Este enfoque, cuando funciona, puede ser una forma muy poderosa de escribir programas. Parte del poder proviene del hecho de que un solo hecho de "qué es" puede usarse para resolver una serie de problemas diferentes que tendrían diferentes componentes de "cómo hacerlo". Como ejemplo, considere la operación `append`, que toma dos listas como argumentos y combina sus elementos para formar una sola lista. En un lenguaje procedimental como Lisp, podríamos definir `append` en términos del constructor de listas básico `cons`, como lo hicimos en Section 2.2.1:

```
(define (append x y) (if (null? x) y (cons (car x) (append (cdr x) y))))
```

Este procedimiento puede considerarse una traducción a Lisp de las siguientes dos reglas, la primera de las cuales cubre el caso en que la primera lista está vacía y la segunda de las cuales maneja el caso de una lista no vacía, que es un `cons` de dos partes:

- Para cualquier lista `y`, la lista vacía y `y se concatenan` para formar `y`.
- Para cualquier `u`, `v`, `y` y `z`, `(cons u v)` y `y se concatenan` para formar `(cons u z)` si `v` y `y se concatenan` para formar `z`.[84]

Usando el procedimiento `append`, podemos responder preguntas como

> Encuentra la `concatenación` de `(a b)` y `(c d)`.

Pero las mismas dos reglas también son suficientes para responder los siguientes tipos de preguntas, que el procedimiento no puede responder:

> Encuentra una lista `y` que se `concatene` con `(a b)` para producir `(a b c d)`.

> Encuentra todas las `x` y `y` que se `concatenan` para formar `(a b c d)`.

En un lenguaje de programación lógica, el programador escribe un "procedimiento" `append` enunciando las dos reglas sobre `append` dadas anteriormente. El conocimiento de "cómo hacerlo" es proporcionado automáticamente por el intérprete para permitir que este único par de reglas se use para responder los tres tipos de preguntas sobre `append`.[85]

Los lenguajes de programación lógica contemporáneos (incluido el que implementamos aquí) tienen deficiencias sustanciales, en el sentido de que sus métodos generales de "cómo hacerlo" pueden llevarlos a bucles infinitos espurios u otros comportamientos indeseables. La programación lógica es un campo activo de investigación en informática.[86]

Anteriormente en este capítulo exploramos la tecnología de implementación de intérpretes y describimos los elementos que son esenciales para un intérprete para un lenguaje similar a Lisp (de hecho, para un intérprete para cualquier lenguaje convencional). Ahora aplicaremos estas ideas para discutir un intérprete para un lenguaje de programación lógica. Llamamos a este lenguaje el *lenguaje de consulta*, porque es muy útil para recuperar información de bases de datos formulando *consultas*, o preguntas, expresadas en el lenguaje. Aunque el lenguaje de consulta es muy diferente de Lisp, encontraremos conveniente describir el lenguaje en términos del mismo marco general que hemos estado usando todo el tiempo: como una colección de elementos primitivos, junto con medios de combinación que nos permiten

combinar elementos simples para crear elementos más complejos y medios de abstracción que nos permiten considerar elementos complejos como unidades conceptuales únicas. Un intérprete para un lenguaje de programación lógica es considerablemente más complejo que un intérprete para un lenguaje como Lisp. Sin embargo, veremos que nuestro intérprete de lenguaje de consulta contiene muchos de los mismos elementos que se encuentran en el intérprete de Section 4.1. En particular, habrá una parte "eval" que clasifica las expresiones según el tipo y una parte "apply" que implementa el mecanismo de abstracción del lenguaje (procedimientos en el caso de Lisp, y *reglas* en el caso de la programación lógica). Además, un papel central lo desempeña en la implementación una estructura de datos de marco, que determina la correspondencia entre los símbolos y sus valores asociados. Un aspecto interesante adicional de nuestra implementación del lenguaje de consulta es que hacemos un uso sustancial de los flujos, que se introdujeron en el Capítulo 3.

Recuperación de Información Deductiva

La programación lógica sobresale en proporcionar interfaces a bases de datos para la recuperación de información. El lenguaje de consulta que implementaremos en este capítulo está diseñado para ser utilizado de esta manera.

Para ilustrar lo que hace el sistema de consulta, mostraremos cómo puede ser utilizado para gestionar la base de datos de registros de personal de Microshaft, una próspera empresa de alta tecnología en el área de Boston. El lenguaje proporciona acceso dirigido por patrones a la información del personal y también puede aprovechar reglas generales para hacer deducciones lógicas.

Una base de datos de muestra

La base de datos de personal de Microshaft contiene *afirmaciones* sobre el personal de la empresa. Aquí está la información sobre Ben Bitdiddle, el mago de la computación residente:

(address (Bitdiddle Ben) (Slumerville (Ridge Road) 10)) (job (Bitdiddle Ben) (computer wizard)) (salary (Bitdiddle Ben) 60000)

Cada afirmación es una lista (en este caso un triple) cuyos elementos pueden ser a su vez listas.

Como mago residente, Ben está a cargo de la división de computación de la

compañía, y supervisa a dos programadores y un técnico. Aquí está la información sobre ellos:

(address (Hacker Alyssa P) (Cambridge (Mass Ave) 78)) (job (Hacker Alyssa P) (computer programmer)) (salary (Hacker Alyssa P) 40000) (supervisor (Hacker Alyssa P) (Bitdiddle Ben))

(address (Fect Cy D) (Cambridge (Ames Street) 3)) (job (Fect Cy D) (computer programmer)) (salary (Fect Cy D) 35000) (supervisor (Fect Cy D) (Bitdiddle Ben))

(address (Tweakit Lem E) (Boston (Bay State Road) 22)) (job (Tweakit Lem E) (computer technician)) (salary (Tweakit Lem E) 25000) (supervisor (Tweakit Lem E) (Bitdiddle Ben))

También hay un programador en prácticas, que es supervisado por Alyssa:

(address (Reasoner Louis) (Slumerville (Pine Tree Road) 80)) (job (Reasoner Louis) (computer programmer trainee)) (salary (Reasoner Louis) 30000) (supervisor (Reasoner Louis) (Hacker Alyssa P))

Todas estas personas están en la división de computación, como lo indica la palabra `computer` como primer elemento en la descripción de su trabajo.

Ben es un empleado de alto nivel. Su supervisor es el pez gordo de la compañía en persona:

(supervisor (Bitdiddle Ben) (Warbucks Oliver)) (address (Warbucks Oliver) (Swellesley (Top Heap Road))) (job (Warbucks Oliver) (administration big wheel)) (salary (Warbucks Oliver) 150000)

Además de la división de computación supervisada por Ben, la compañía tiene una división de contabilidad, que consiste en un contador jefe y su asistente:

(address (Scrooge Eben) (Weston (Shady Lane) 10)) (job (Scrooge Eben) (accounting chief accountant)) (salary (Scrooge Eben) 75000) (supervisor (Scrooge Eben) (Warbucks Oliver))

(address (Cratchet Robert) (Allston (N Harvard Street) 16)) (job (Cratchet Robert) (accounting scrivener)) (salary (Cratchet Robert) 18000) (supervisor (Cratchet Robert) (Scrooge Eben))

También hay una secretaria para el pez gordo:

(address (Aull DeWitt) (Slumerville (Onion Square) 5)) (job (Aull DeWitt)

(administration secretary)) (salary (Aull DeWitt) 25000) (supervisor (Aull DeWitt) (Warbucks Oliver))

La base de datos también contiene afirmaciones sobre qué tipos de trabajos pueden ser realizados por personas que ocupan otros tipos de trabajos. Por ejemplo, un mago de la computación puede hacer los trabajos tanto de un programador de computadoras como de un técnico de computadoras:

(can-do-job (computer wizard) (computer programmer)) (can-do-job (computer wizard) (computer technician))

Un programador de computadoras podría reemplazar a un aprendiz:

(can-do-job (computer programmer) (computer programmer trainee))

Además, como es bien sabido,

(can-do-job (administration secretary) (administration big wheel))

Consultas simples

El lenguaje de consulta permite a los usuarios recuperar información de la base de datos planteando consultas en respuesta al prompt del sistema. Por ejemplo, para encontrar a todos los programadores de computadoras se puede decir:

;;; Entrada de consulta: (job ?x (computer programmer))

El sistema responderá con los siguientes elementos:

;;; Resultados de la consulta: (job (Hacker Alyssa P) (computer programmer)) (job (Fect Cy D) (computer programmer))

La consulta de entrada especifica que estamos buscando entradas en la base de datos que coincidan con un cierto *patrón*. En este ejemplo, el patrón especifica entradas que consisten en tres elementos, de los cuales el primero es el símbolo literal `job`, el segundo puede ser cualquier cosa, y el tercero es la lista literal `(computer programmer)`. El "cualquier cosa" que puede ser el segundo elemento en la lista coincidente se especifica mediante una *variable de patrón*, `?x`. La forma general de una variable de patrón es un símbolo, tomado como el nombre de la variable, precedido por un signo de interrogación. Veremos más adelante por qué es útil especificar nombres para las variables de patrón en lugar de simplemente poner `?` en los patrones para representar "cualquier cosa". El sistema responde a una consulta simple mostrando todas las entradas en la base de datos que coinciden con el patrón

especificado.

Un patrón puede tener más de una variable. Por ejemplo, la consulta

(address ?x ?y)

listará todas las direcciones de los empleados.

Un patrón puede no tener variables, en cuyo caso la consulta simplemente determina si ese patrón es una entrada en la base de datos. Si es así, habrá una coincidencia; si no, no habrá ninguna.

La misma variable de patrón puede aparecer más de una vez en una consulta, especificando que el mismo "cualquier cosa" debe aparecer en cada posición. Es por esto que las variables tienen nombres. Por ejemplo,

(supervisor ?x ?x)

encuentra a todas las personas que se supervisan a sí mismas (aunque no hay tales afirmaciones en nuestra base de datos de muestra).

La consulta

(job ?x (computer ?type))

coincide con todas las entradas de trabajo cuyo tercer elemento es una lista de dos elementos cuyo primer elemento es `computer`:

(job (Bitdiddle Ben) (computer wizard)) (job (Hacker Alyssa P) (computer programmer)) (job (Fect Cy D) (computer programmer)) (job (Tweakit Lem E) (computer technician))

Este mismo patrón *no* coincide con

(job (Reasoner Louis) (computer programmer trainee))

porque el tercer elemento en la entrada es una lista de tres elementos, y el tercer elemento del patrón especifica que debe haber dos elementos. Si quisiéramos cambiar el patrón para que el tercer elemento pudiera ser cualquier lista que comience con `computer`, podríamos especificar[87]

(job ?x (computer . ?type))

Por ejemplo,

(computer . ?type)

coincide con los datos

(computer programmer trainee)

con `?type` como la lista `(programmer trainee)`. También coincide con los datos

(computer programmer)

con `?type` como la lista `(programmer)`, y coincide con los datos

(computer)

con `?type` como la lista vacía `()`.

Podemos describir el procesamiento de consultas simples del lenguaje de consulta de la siguiente manera:

- El sistema encuentra todas las asignaciones a las variables del patrón de consulta que *satisfacen* el patrón—es decir, todos los conjuntos de valores para las variables tales que si las variables del patrón se *instancian con* (reemplazan por) los valores, el resultado está en la base de datos.
- El sistema responde a la consulta listando todas las instancias del patrón de consulta con las asignaciones de variables que lo satisfacen.

Nótese que si el patrón no tiene variables, la consulta se reduce a determinar si ese patrón está en la base de datos. Si es así, la asignación vacía, que no asigna valores a las variables, satisface ese patrón para esa base de datos.

> **Ejercicio 4.55:** Dé consultas simples que recuperen la siguiente información de la base de datos:
>
> 1. todas las personas supervisadas por Ben Bitdiddle;
> 2. los nombres y trabajos de todas las personas de la división de contabilidad;
> 3. los nombres y direcciones de todas las personas que viven en Slumerville.

Consultas compuestas

Las consultas simples forman las operaciones primitivas del lenguaje de consulta. Para formar operaciones compuestas, el lenguaje de consulta proporciona medios de combinación. Una cosa que hace que el lenguaje de

consulta sea un lenguaje de programación lógica es que los medios de combinación reflejan los medios de combinación utilizados en la formación de expresiones lógicas: `and`, `or` y `not`. (Aquí `and`, `or` y `not` no son las primitivas de Lisp, sino operaciones incorporadas en el lenguaje de consulta.)

Podemos usar `and` de la siguiente manera para encontrar las direcciones de todos los programadores de computadoras:

(and (job ?person (computer programmer)) (address ?person ?where))

El resultado es:

(and (job (Hacker Alyssa P) (computer programmer)) (address (Hacker Alyssa P) (Cambridge (Mass Ave) 78))) (and (job (Fect Cy D) (computer programmer)) (address (Fect Cy D) (Cambridge (Ames Street) 3)))

En general,

(and < *consulta* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$ < *consulta* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$ … < *consulta* $\color{SchemeDark}_{\hbox{\ttfamily \itshape\scriptsize n}}\rangle$)

se satisface con todos los conjuntos de valores para las variables de patrón que satisfacen simultáneamente ⟨*consulta*$_1$⟩ … ⟨*consulta*$_n$⟩.

Al igual que con las consultas simples, el sistema procesa una consulta compuesta encontrando todas las asignaciones a las variables del patrón que satisfacen la consulta, y luego mostrando instancias de la consulta con esos valores.

Otro medio de construir consultas compuestas es a través de `or`. Por ejemplo,

(or (supervisor ?x (Bitdiddle Ben)) (supervisor ?x (Hacker Alyssa P)))

encontrará a todos los empleados supervisados por Ben Bitdiddle o Alyssa P. Hacker:

(or (supervisor (Hacker Alyssa P) (Bitdiddle Ben)) (supervisor (Hacker Alyssa P) (Hacker Alyssa P))) (or (supervisor (Fect Cy D) (Bitdiddle Ben)) (supervisor (Fect Cy D) (Hacker Alyssa P))) (or (supervisor (Tweakit Lem E) (Bitdiddle Ben)) (supervisor (Tweakit Lem E) (Hacker Alyssa P))) (or (supervisor (Reasoner Louis) (Bitdiddle Ben)) (supervisor (Reasoner Louis) (Hacker Alyssa P)))

En general,

(or < *consulta* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle $ < *consulta* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle $ … < *consulta* $\color{SchemeDark}_{\hbox{\ttfamily\itshape\scriptsize n}}\rangle$)

se satisface con todos los conjuntos de valores para las variables de patrón que satisfacen al menos uno de ⟨*consulta*$_1$⟩ … ⟨*consulta*$_n$⟩.

Las consultas compuestas también pueden formarse con `not`. Por ejemplo,

(and (supervisor ?x (Bitdiddle Ben)) (not (job ?x (computer programmer))))

encuentra a todas las personas supervisadas por Ben Bitdiddle que no son programadores de computadoras. En general,

(not < *consulta* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$)

se satisface con todas las asignaciones a las variables del patrón que no satisfacen ⟨*consulta*$_1$⟩.[88]

La forma de combinación final se llama `lisp/value`. Cuando `lisp/value` es el primer elemento de un patrón, especifica que el siguiente elemento es un predicado de Lisp que se aplicará al resto de los elementos (instanciados) como argumentos. En general,

(lisp-value < *predicado* > < *arg* $\color{SchemeDark}_{\hbox{\ttfamily \scriptsize 1}}\rangle$ … < *arg* $\color{SchemeDark}_{\hbox{\ttfamily \itshape\scriptsize n}}\rangle$)

será satisfecho por las asignaciones a las variables del patrón para las cuales el ⟨*predicado*⟩ aplicado a los ⟨*arg*$_1$⟩ … ⟨*arg*$_n$⟩ instanciados es verdadero. Por ejemplo, para encontrar a todas las personas cuyo salario es mayor que $30,000 podríamos escribir[89]

(and (salary ?person ?amount) (lisp-value > ?amount 30000))

Ejercicio 4.56: Formule consultas compuestas que recuperen la siguiente información:

1. los nombres de todas las personas que son supervisadas por Ben Bitdiddle, junto con sus direcciones;

2. todas las personas cuyo salario es menor que el de Ben Bitdiddle, junto con su salario y el salario de Ben Bitdiddle;

3. todas las personas que son supervisadas por alguien que no está en la división de computación, junto con el nombre y el trabajo del supervisor.

Reglas

Además de las consultas primitivas y compuestas, el lenguaje de consulta proporciona medios para abstraer consultas. Estos están dados por *reglas*. La regla

(rule (lives-near ?person-1 ?person-2) (and (address ?person-1 (?town . ?rest-1)) (address ?person-2 (?town . ?rest-2)) (not (same ?person-1 ?person-2))))

especifica que dos personas viven cerca una de la otra si viven en la misma ciudad. La cláusula final `not` evita que la regla diga que todas las personas viven cerca de sí mismas. La relación `same` se define mediante una regla muy simple:[90]

(rule (same ?x ?x))

La siguiente regla declara que una persona es un "pez gordo" en una organización si supervisa a alguien que a su vez es un supervisor:

(rule (wheel ?person) (and (supervisor ?middle-manager ?person) (supervisor ?x ?middle-manager)))

La forma general de una regla es

(rule ⟨ *conclusión* ⟩ ⟨ *cuerpo* ⟩)

donde ⟨*conclusión*⟩ es un patrón y ⟨*cuerpo*⟩ es cualquier consulta.[91] Podemos pensar en una regla como si representara un conjunto grande (incluso infinito) de afirmaciones, a saber, todas las instancias de la conclusión de la regla con asignaciones de variables que satisfacen el cuerpo de la regla. Cuando describimos consultas simples (patrones), dijimos que una asignación a variables satisface un patrón si el patrón instanciado está en la base de datos. Pero el patrón no necesita estar explícitamente en la base de datos como una afirmación. Puede ser una afirmación implícita implicada por una regla. Por ejemplo, la consulta

(lives-near ?x (Bitdiddle Ben))

resulta en

(lives-near (Reasoner Louis) (Bitdiddle Ben)) (lives-near (Aull DeWitt) (Bitdiddle Ben))

Para encontrar a todos los programadores de computadoras que viven cerca de Ben Bitdiddle, podemos preguntar

(and (job ?x (computer programmer)) (lives-near ?x (Bitdiddle Ben)))

Al igual que en el caso de los procedimientos compuestos, las reglas pueden usarse como partes de otras reglas (como vimos con la regla `lives/near` anterior) o incluso definirse recursivamente. Por ejemplo, la regla

(rule (outranked-by ?staff-person ?boss) (or (supervisor ?staff-person ?boss) (and (supervisor ?staff-person ?middle-manager) (outranked-by ?middle-manager ?boss))))

dice que un miembro del personal está subordinado a un jefe en la organización si el jefe es el supervisor de la persona o (recursivamente) si el supervisor de la persona está subordinado al jefe.

Ejercicio 4.57: Defina una regla que diga que la persona 1 puede reemplazar a la persona 2 si la persona 1 hace el mismo trabajo que la persona 2 o alguien que hace el trabajo de la persona 1 también puede hacer el trabajo de la persona 2, y si la persona 1 y la persona 2 no son la misma persona. Usando su regla, dé consultas que encuentren lo siguiente:

1. todas las personas que pueden reemplazar a Cy D. Fect;

2. todas las personas que pueden reemplazar a alguien a quien se le paga más que a ellas, junto con los dos salarios.

Ejercicio 4.58: Defina una regla que diga que una persona es un "pez gordo" en una división si la persona trabaja en la división pero no tiene un supervisor que trabaje en la división.

Ejercicio 4.59: Ben Bitdiddle se ha perdido una reunión de más. Temiendo que su hábito de olvidar las reuniones pueda costarle el trabajo, Ben decide hacer algo al respecto. Agrega todas las reuniones semanales de la empresa a la base de datos de Microshaft afirmando lo siguiente:

(meeting accounting (Monday 9am)) (meeting administration (Monday 10am)) (meeting computer (Wednesday 3pm)) (meeting administration (Friday 1pm))

Cada una de las afirmaciones anteriores es para una reunión de toda una división. Ben también agrega una entrada para la reunión de toda la empresa que abarca todas las divisiones. Todos los empleados de la empresa asisten a esta reunión.

(meeting whole-company (Wednesday 4pm))

1. El viernes por la mañana, Ben quiere consultar la base de datos para todas las reuniones que ocurren ese día. ¿Qué consulta debería usar?

2. Alyssa P. Hacker no está impresionada. Cree que sería mucho más útil poder pedir sus reuniones especificando su nombre. Así que diseña una regla que dice que las reuniones de una persona incluyen todas las reuniones de `whole/company` más todas las reuniones de la división de esa persona. Complete el cuerpo de la regla de Alyssa.

   (rule (meeting-time ?person ?day-and-time) < *rule-body* >)

3. Alyssa llega al trabajo el miércoles por la mañana y se pregunta a qué reuniones tiene que asistir ese día. Habiendo definido la regla anterior, ¿qué consulta debería hacer para averiguarlo?

Ejercicio 4.60: Al dar la consulta

(lives-near ?person (Hacker Alyssa P))

Alyssa P. Hacker es capaz de encontrar personas que viven cerca de ella, con las que puede ir al trabajo. Por otro lado, cuando intenta encontrar todos los pares de personas que viven cerca una de la otra consultando

(lives-near ?person-1 ?person-2)

ella nota que cada par de personas que viven cerca una de la otra aparece dos veces; por ejemplo,

(lives-near (Hacker Alyssa P) (Fect Cy D)) (lives-near (Fect Cy D) (Hacker Alyssa P))

¿Por qué sucede esto? ¿Hay alguna forma de encontrar una lista de personas que viven cerca una de la otra, en la que cada par aparezca solo una vez? Explique.

Lógica como programas

Podemos considerar una regla como una especie de implicación lógica: *Si* una

asignación de valores a las variables del patrón satisface el cuerpo, *entonces* satisface la conclusión. En consecuencia, podemos considerar que el lenguaje de consulta tiene la capacidad de realizar *deducciones lógicas* basadas en las reglas. Como ejemplo, considere la operación `append` descrita al principio de Section 4.4. Como dijimos, `append` puede caracterizarse por las siguientes dos reglas:

- Para cualquier lista `y`, la lista vacía y `y se concatenan` para formar `y`.
- Para cualquier `u`, `v`, `y` y `z`, `(cons u v)` y `y se concatenan` para formar `(cons u z)` si `v` y `y se concatenan` para formar `z`.

Para expresar esto en nuestro lenguaje de consulta, definimos dos reglas para una relación

(append-to-form x y z)

que podemos interpretar como "`x` y `y se concatenan` para formar `z`":

(rule (append-to-form () ?y ?y)) (rule (append-to-form (?u . ?v) ?y (?u . ?z)) (append-to-form ?v ?y ?z))

La primera regla no tiene cuerpo, lo que significa que la conclusión es válida para cualquier valor de `?y`. Nótese cómo la segunda regla hace uso de la notación de cola punteada para nombrar el `car` y el `cdr` de una lista.

Dadas estas dos reglas, podemos formular consultas que calculan la `concatenación` de dos listas:

;;; Entrada de consulta: (append-to-form (a b) (c d) ?z) *;;; Resultados de la consulta:* (append-to-form (a b) (c d) (a b c d))

Lo que es más sorprendente, podemos usar las mismas reglas para hacer la pregunta "¿Qué lista, cuando se `concatena` con `(a b)`, produce `(a b c d)`?" Esto se hace de la siguiente manera:

;;; Entrada de consulta: (append-to-form (a b) ?y (a b c d)) *;;; Resultados de la consulta:* (append-to-form (a b) (c d) (a b c d))

También podemos pedir todos los pares de listas que se `concatenan` para formar `(a b c d)`:

;;; Entrada de consulta: (append-to-form ?x ?y (a b c d)) *;;; Resultados de la consulta:* (append-to-form () (a b c d) (a b c d)) (append-to-form (a) (b c d) (a b c d)) (append-to-form (a b) (c d) (a b c d)) (append-to-form (a b c) (d) (a b c d)) (append-to-form (a b c d) () (a b c d))

El sistema de consulta puede parecer exhibir bastante inteligencia al usar las reglas para deducir las respuestas a las consultas anteriores. En realidad, como veremos en la siguiente sección, el sistema sigue un algoritmo bien determinado al desentrañar las reglas. Desafortunadamente, aunque el sistema funciona de manera impresionante en el caso de `append`, los métodos generales pueden fallar en casos más complejos, como veremos en Section 4.4.3.

Ejercicio 4.61: Las siguientes reglas implementan una relación `next/to` que encuentra elementos adyacentes de una lista:

(rule (?x next-to ?y in (?x ?y . ?u))) (rule (?x next-to ?y in (?v . ?z)) (?x next-to ?y in ?z))

¿Cuál será la respuesta a las siguientes consultas?

(?x next-to ?y in (1 (2 3) 4)) (?x next-to 1 in (2 1 3 1))

Ejercicio 4.62: Defina reglas para implementar la operación `last/pair` del Ejercicio 2.17, que devuelve una lista que contiene el último elemento de una lista no vacía. Compruebe sus reglas en consultas como `(last/pair (3) ?x)`, `(last/pair (1 2 3) ?x)` y `(last/pair (2 ?x) (3))`. ¿Funcionan sus reglas correctamente en consultas como `(last/pair ?x (3))`?

Ejercicio 4.63: La siguiente base de datos (ver Génesis 4) traza la genealogía de los descendientes de Ada hasta Adán, a través de Caín:

(son Adam Cain) (son Cain Enoch) (son Enoch Irad) (son Irad Mehujael) (son Mehujael Methushael) (son Methushael Lamech) (wife Lamech Ada) (son Ada Jabal) (son Ada Jubal)

Formule reglas como "Si *S* es el hijo de *f*, y *f* es el hijo de *G*, entonces *S* es el nieto de *G*" y "Si *W* es la esposa de *M*, y *S* es el hijo de *W*, entonces *S* es el hijo de *M*" (que supuestamente era más cierto en los tiempos bíblicos que hoy) que permitirán que el sistema de consulta encuentre al nieto de Caín; los hijos de Lamec; los nietos de Matusalén. (Ver Ejercicio 4.69 para algunas reglas para deducir relaciones más complicadas.)

Cómo funciona el sistema de consulta

El Section 4.4.4 presentaremos una implementación del intérprete de consultas como una colección de procedimientos. En esta sección, ofrecemos una visión general que explica la estructura general del sistema

independientemente de los detalles de implementación de bajo nivel. Después de describir la implementación del intérprete, estaremos en condiciones de comprender algunas de sus limitaciones y algunas de las formas sutiles en que las operaciones lógicas del lenguaje de consulta difieren de las operaciones de la lógica matemática.

Debería ser evidente que el evaluador de consultas debe realizar algún tipo de búsqueda para hacer coincidir las consultas con los hechos y las reglas de la base de datos. Una forma de hacerlo sería implementar el sistema de consultas como un programa no determinista, usando el evaluador `amb` de Section 4.3 (ver Ejercicio 4.78). Otra posibilidad es gestionar la búsqueda con la ayuda de flujos. Nuestra implementación sigue este segundo enfoque.

El sistema de consulta se organiza en torno a dos operaciones centrales llamadas *coincidencia de patrones* y *unificación*. Primero describimos la coincidencia de patrones y explicamos cómo esta operación, junto con la organización de la información en términos de flujos de marcos, nos permite implementar tanto consultas simples como compuestas. A continuación, discutimos la unificación, una generalización de la coincidencia de patrones necesaria para implementar reglas. Finalmente, mostramos cómo todo el intérprete de consultas encaja a través de un procedimiento que clasifica las expresiones de una manera análoga a la forma en que `eval` clasifica las expresiones para el intérprete descrito en Section 4.1.

Coincidencia de patrones

Un *coincidente de patrones* es un programa que prueba si algún dato se ajusta a un patrón especificado. Por ejemplo, la lista de datos `((a b) c (a b))` coincide con el patrón `(?x c ?x)` con la variable de patrón `?x` ligada a `(a b)`. La misma lista de datos coincide con el patrón `(?x ?y ?z)` con `?x` y `?z` ambos ligados a `(a b)` y `?y` ligado a `c`. También coincide con el patrón `((?x ?y) c (?x ?y))` con `?x` ligado a `a` y `?y` ligado a `b`. Sin embargo, no coincide con el patrón `(?x a ?y)`, ya que ese patrón especifica una lista cuyo segundo elemento es el símbolo `a`.

El coincidente de patrones utilizado por el sistema de consulta toma como entrada un patrón, un dato y un *marco* que especifica ligaduras para varias variables de patrón. Comprueba si el dato coincide con el patrón de una manera que sea consistente con las ligaduras ya existentes en el marco. Si es así, devuelve el marco dado aumentado con cualquier ligadura que pueda haber sido determinada por la coincidencia. De lo contrario, indica que la coincidencia ha fallado.

Por ejemplo, usar el patrón `(?x ?y ?x)` para coincidir con `(a b a)` dado un

marco vacío devolverá un marco que especifica que `?x` está ligado a `a` y `?y` está ligado a `b`. Intentar la coincidencia con el mismo patrón, el mismo dato y un marco que especifica que `?y` está ligado a `a` fallará. Intentar la coincidencia con el mismo patrón, el mismo dato y un marco en el que `?y` está ligado a `b` y `?x` no está ligado devolverá el marco dado aumentado con una ligadura de `?x` a `a`.

El coincidente de patrones es todo el mecanismo que se necesita para procesar consultas simples que no involucran reglas. Por ejemplo, para procesar la consulta

(job ?x (computer programmer))

recorremos todas las afirmaciones en la base de datos y seleccionamos aquellas que coinciden con el patrón con respecto a un marco inicialmente vacío. Por cada coincidencia que encontramos, usamos el marco devuelto por la coincidencia para instanciar el patrón con un valor para `?x`.

Flujos de marcos

La prueba de patrones contra marcos se organiza mediante el uso de flujos. Dado un solo marco, el proceso de coincidencia recorre las entradas de la base de datos una por una. Para cada entrada de la base de datos, el coincidente genera o bien un símbolo especial que indica que la coincidencia ha fallado o una extensión al marco. Los resultados de todas las entradas de la base de datos se recopilan en un flujo, que se pasa a través de un filtro para eliminar los fallos. El resultado es un flujo de todos los marcos que extienden el marco dado mediante una coincidencia con alguna afirmación en la base de datos.[92]

En nuestro sistema, una consulta toma un flujo de entrada de marcos y realiza la operación de coincidencia anterior para cada marco en el flujo, como se indica en la Figura 4.4. Es decir, para cada marco en el flujo de entrada, la consulta genera un nuevo flujo que consiste en todas las extensiones a ese marco mediante coincidencias con afirmaciones en la base de datos. Todos estos flujos se combinan para formar un gran flujo, que contiene todas las extensiones posibles de cada marco en el flujo de entrada. Este flujo es la salida de la consulta.

Para responder a una consulta simple, usamos la consulta con un flujo de entrada que consiste en un único marco vacío. El flujo de salida resultante contiene todas las extensiones al marco vacío (es decir, todas las respuestas a nuestra consulta). Este flujo de marcos se usa luego para generar un flujo de copias del patrón de consulta original con las variables instanciadas por los

valores de cada marco, y este es el flujo que finalmente se imprime.

Figura 4.4: Una consulta procesa un flujo de marcos.

Consultas compuestas

La verdadera elegancia de la implementación del flujo de marcos es evidente cuando tratamos con consultas compuestas. El procesamiento de consultas compuestas hace uso de la capacidad de nuestro comparador para exigir que una coincidencia sea consistente con un marco especificado. Por ejemplo, para manejar el `y` de dos consultas, como

(and (can-do-job ?x (computer programmer trainee)) (job ?person ?x))

(informalmente, "Encontrar a todas las personas que pueden hacer el trabajo de un programador de computadoras en prácticas"), primero encontramos todas las entradas que coinciden con el patrón

(can-do-job ?x (computer programmer trainee))

Figura 4.5: La combinación `y` de dos consultas se produce operando sobre el flujo de marcos en serie.

Esto produce un flujo de marcos, cada uno de los cuales contiene una ligadura para `?x`. Luego, para cada marco en el flujo, encontramos todas las entradas que coinciden

(job ?person ?x)

de una manera que sea consistente con la ligadura dada para `?x`. Cada una de estas coincidencias producirá un marco que contiene ligaduras para `?x` y `?person`. El `y` de dos consultas puede verse como una combinación en serie de las dos consultas componentes, como se muestra en la Figura 4.5. Los marcos que pasan a través del primer filtro de consulta son filtrados y extendidos aún más por la segunda consulta.

Figura 4.6: La combinación `or` de dos consultas se produce operando sobre el flujo de marcos en paralelo y fusionando los resultados.

La Figura 4.6 muestra el método análogo para calcular el `o` de dos consultas como una combinación paralela de las dos consultas componentes. El flujo de entrada de marcos se extiende por separado por cada consulta. Los dos flujos resultantes se fusionan para producir el flujo de salida final.

Incluso a partir de esta descripción de alto nivel, es evidente que el procesamiento de consultas compuestas puede ser lento. Por ejemplo, dado que una consulta puede producir más de un marco de salida por cada marco de entrada, y cada consulta en un `y` obtiene sus marcos de entrada de la consulta anterior, una consulta `y` podría, en el peor de los casos, tener que realizar un número de coincidencias que es exponencial en el número de consultas (ver Ejercicio 4.76).[93] Aunque los sistemas para manejar solo consultas simples son bastante prácticos, tratar con consultas complejas es extremadamente difícil.[94]

Desde el punto de vista del flujo de marcos, el `no` de alguna consulta actúa como un filtro que elimina todos los marcos para los que la consulta puede ser satisfecha. Por ejemplo, dado el patrón

(not (job ?x (computer programmer)))

intentamos, para cada marco en el flujo de entrada, producir marcos de extensión que satisfagan `(job ?x (computer programmer))`. Eliminamos del flujo de entrada todos los marcos para los que existen tales extensiones. El resultado es un flujo que consiste solo en aquellos marcos en los que la ligadura para `?x` no satisface la restricción de que `?x` es un programador de computadoras. Por ejemplo, al procesar la consulta

(and (supervisor ?x ?y) (not (job ?x (computer programmer))))

la primera cláusula generará marcos con ligaduras para `?x` y `?y`. La cláusula `not` luego los filtrará eliminando todos los marcos en los que la ligadura para `?x` satisfaga la restricción de que `?x` es un programador de computadoras.[95]

La forma especial `lisp/value` se implementa como un filtro similar en los flujos de marcos. Usamos cada marco del flujo para instanciar cualquier variable del patrón, luego aplicamos el predicado de Lisp. Eliminamos del flujo de entrada todos los marcos para los que el predicado falla.

Unificación

Para manejar las reglas en el lenguaje de consulta, debemos ser capaces de encontrar las reglas cuyas conclusiones coincidan con un patrón de consulta dado. Las conclusiones de las reglas son como las aserciones, excepto que

pueden contener variables, por lo que necesitaremos una generalización de la coincidencia de patrones—llamada *unificación*—en la que tanto el "patrón" como el "dato" pueden contener variables.

Un unificador toma dos patrones, cada uno conteniendo constantes y variables, y determina si es posible asignar valores a las variables que harán que los dos patrones sean iguales. Si es así, devuelve un marco que contiene estas ligaduras. Por ejemplo, unificar `(?x a ?y)` y `(?y ?z a)` especificará un marco en el que `?x`, `?y` y `?z` deben estar todos ligados a `a`. Por otro lado, unificar `(?x ?y a)` y `(?x b ?y)` fallará, porque no hay ningún valor para `?y` que pueda hacer que los dos patrones sean iguales. (Para que los segundos elementos de los patrones sean iguales, `?y` tendría que ser `b`; sin embargo, para que los terceros elementos sean iguales, `?y` tendría que ser `a`.) El unificador utilizado en el sistema de consulta, al igual que el comparador de patrones, toma un marco como entrada y realiza unificaciones que son consistentes con este marco.

El algoritmo de unificación es la parte técnicamente más difícil del sistema de consulta. Con patrones complejos, realizar la unificación puede parecer que requiere deducción. Para unificar `(?x ?x)` y `((a ?y c) (a b ?z))`, por ejemplo, el algoritmo debe inferir que `?x` debe ser `(a b c)`, `?y` debe ser `b`, y `?z` debe ser `c`. Podemos pensar en este proceso como la resolución de un conjunto de ecuaciones entre los componentes del patrón. En general, estas son ecuaciones simultáneas, que pueden requerir una manipulación sustancial para resolverse.[96] Por ejemplo, unificar `(?x ?x)` y `((a ?y c) (a b ?z))` puede considerarse como la especificación de las ecuaciones simultáneas

?x = (a ?y c) ?x = (a b ?z)

Estas ecuaciones implican que

(a ?y c) = (a b ?z)

lo que a su vez implica que

a = a, ?y = b, c = ?z,

y por lo tanto que

?x = (a b c)

En una coincidencia de patrones exitosa, todas las variables del patrón quedan ligadas, y los valores a los que están ligadas contienen solo constantes. Esto también es cierto para todos los ejemplos de unificación que

hemos visto hasta ahora. En general, sin embargo, una unificación exitosa puede no determinar completamente los valores de las variables; algunas variables pueden permanecer sin ligar y otras pueden estar ligadas a valores que contienen variables.

Considere la unificación de `(?x a)` y `((b ?y) ?z)`. Podemos deducir que `?x = (b ?y)` y `a = ?z`, pero no podemos resolver más para `?x` o `?y`. La unificación no falla, ya que ciertamente es posible hacer que los dos patrones sean iguales asignando valores a `?x` y `?y`. Como esta coincidencia de ninguna manera restringe los valores que `?y` puede tomar, no se pone ninguna ligadura para `?y` en el marco resultante. Sin embargo, la coincidencia sí restringe el valor de `?x`. Sea cual sea el valor que tenga `?y`, `?x` debe ser `(b ?y)`. Por lo tanto, se pone en el marco una ligadura de `?x` al patrón `(b ?y)`. Si posteriormente se determina un valor para `?y` y se agrega al marco (mediante una coincidencia de patrones o una unificación que se requiere que sea consistente con este marco), el `?x` previamente ligado se referirá a este valor.[97]

Aplicando reglas

La unificación es la clave del componente del sistema de consulta que hace inferencias a partir de las reglas. Para ver cómo se logra esto, considere procesar una consulta que implique la aplicación de una regla, como

(lives-near ?x (Hacker Alyssa P))

Para procesar esta consulta, primero usamos el procedimiento de coincidencia de patrones ordinario descrito anteriormente para ver si hay alguna aserción en la base de datos que coincida con este patrón. (No habrá ninguna en este caso, ya que nuestra base de datos no incluye ninguna aserción directa sobre quién vive cerca de quién.) El siguiente paso es intentar unificar el patrón de la consulta con la conclusión de cada regla. Encontramos que el patrón se unifica con la conclusión de la regla

(rule (lives-near ?person-1 ?person-2) (and (address ?person-1 (?town . ?rest-1)) (address ?person-2 (?town . ?rest-2)) (not (same ?person-1 ?person-2))))

resultando en un marco que especifica que `?person/2` está ligado a `(Hacker Alyssa P)` y que `?x` debería estar ligado a (tener el mismo valor que) `?person/1`. Ahora, en relación con este marco, evaluamos la consulta compuesta dada por el cuerpo de la regla. Las coincidencias exitosas extenderán este marco proporcionando una ligadura para `?person/1`, y en consecuencia un valor para `?x`, que podemos usar para instanciar el patrón

de la consulta original.

En general, el evaluador de consultas utiliza el siguiente método para aplicar una regla al tratar de establecer un patrón de consulta en un marco que especifica ligaduras para algunas de las variables del patrón:

- Unificar la consulta con la conclusión de la regla para formar, si tiene éxito, una extensión del marco original.
- En relación con el marco extendido, evaluar la consulta formada por el cuerpo de la regla.

Nótese cuán similar es esto al método para aplicar un procedimiento en el evaluador `eval/apply` para Lisp:

- Ligar los parámetros del procedimiento a sus argumentos para formar un marco que extienda el entorno original del procedimiento.
- En relación con el entorno extendido, evaluar la expresión formada por el cuerpo del procedimiento.

La similitud entre los dos evaluadores no debería sorprender. Así como las definiciones de procedimientos son los medios de abstracción en Lisp, las definiciones de reglas son los medios de abstracción en el lenguaje de consulta. En cada caso, desenrollamos la abstracción creando las ligaduras apropiadas y evaluando el cuerpo de la regla o procedimiento en relación con estas.

Consultas simples

Vimos anteriormente en esta sección cómo evaluar consultas simples en ausencia de reglas. Ahora que hemos visto cómo aplicar reglas, podemos describir cómo evaluar consultas simples usando tanto reglas como aserciones.

Dado el patrón de consulta y un flujo de marcos, producimos, para cada marco en el flujo de entrada, dos flujos:

- un flujo de marcos extendidos obtenidos al hacer coincidir el patrón con todas las afirmaciones de la base de datos (usando el comparador de patrones), y
- un flujo de marcos extendidos obtenidos al aplicar todas las reglas posibles (usando el unificador).[98]

Añadir estos dos flujos produce un flujo que consiste en todas las formas en que el patrón dado puede ser satisfecho de manera consistente con el marco original. Estos flujos (uno por cada marco en el flujo de entrada) se combinan ahora para formar un gran flujo, que por lo tanto consiste en todas las formas en que cualquiera de los marcos en el flujo de entrada original puede ser extendido para producir una coincidencia con el patrón dado.

El evaluador de consultas y el bucle del controlador

A pesar de la complejidad de las operaciones de coincidencia subyacentes, el sistema está organizado de manera muy similar a un evaluador para cualquier lenguaje. El procedimiento que coordina las operaciones de coincidencia se llama `qeval`, y desempeña un papel análogo al del procedimiento `eval` para Lisp. `qeval` toma como entradas una consulta y un flujo de marcos. Su salida es un flujo de marcos, correspondiente a coincidencias exitosas con el patrón de la consulta, que extienden algún marco en el flujo de entrada, como se indica en la Figura 4.4. Al igual que `eval`, `qeval` clasifica los diferentes tipos de expresiones (consultas) y despacha a un procedimiento apropiado para cada uno. Hay un procedimiento para cada forma especial (`and`, `or`, `not`, y `lisp/value`) y uno para consultas simples.

El bucle del controlador, que es análogo al procedimiento `driver/loop` de los otros evaluadores de este capítulo, lee las consultas desde el terminal. Para cada consulta, llama a `qeval` con la consulta y un flujo que consiste en un único marco vacío. Esto producirá el flujo de todas las posibles coincidencias (todas las posibles extensiones al marco vacío). Para cada marco en el flujo resultante, instancia la consulta original usando los valores de las variables encontrados en el marco. Este flujo de consultas instanciadas se imprime entonces.[99]

El controlador también comprueba el comando especial `assert!`, que indica que la entrada no es una consulta sino una aserción o regla que se agregará a la base de datos. Por ejemplo,

(assert! (job (Bitdiddle Ben) (computer wizard))) (assert! (rule (wheel ? person) (and (supervisor ?middle-manager ?person) (supervisor ?x ?middle-manager))))

¿Es la programación lógica, lógica matemática?

Los medios de combinación utilizados en el lenguaje de consulta pueden parecer a primera vista idénticos a las operaciones `and`, `or` y `not` de la lógica

matemática, y la aplicación de las reglas del lenguaje de consulta se logra de hecho a través de un método legítimo de inferencia.[100] Esta identificación del lenguaje de consulta con la lógica matemática no es realmente válida, sin embargo, porque el lenguaje de consulta proporciona una *estructura de control* que interpreta las declaraciones lógicas de manera procedimental. A menudo podemos aprovechar esta estructura de control. Por ejemplo, para encontrar a todos los supervisores de programadores podríamos formular una consulta en cualquiera de las dos formas lógicamente equivalentes:

(and (job ?x (computer programmer)) (supervisor ?x ?y))

o

(and (supervisor ?x ?y) (job ?x (computer programmer)))

Si una empresa tiene muchos más supervisores que programadores (el caso habitual), es mejor usar la primera forma en lugar de la segunda porque la base de datos debe ser escaneada por cada resultado intermedio (marco) producido por la primera cláusula del `y`.

El objetivo de la programación lógica es proporcionar al programador técnicas para descomponer un problema computacional en dos problemas separados: "qué" se va a calcular y "cómo" se debe calcular. Esto se logra seleccionando un subconjunto de las declaraciones de la lógica matemática que sea lo suficientemente poderoso como para poder describir cualquier cosa que se quiera calcular, pero lo suficientemente débil como para tener una interpretación procedimental controlable. La intención aquí es que, por un lado, un programa especificado en un lenguaje de programación lógica debe ser un programa efectivo que pueda ser llevado a cabo por una computadora. El control ("cómo" calcular) se efectúa usando el orden de evaluación del lenguaje. Deberíamos poder organizar el orden de las cláusulas y el orden de los subobjetivos dentro de cada cláusula para que el cálculo se realice en un orden considerado efectivo y eficiente. Al mismo tiempo, deberíamos poder ver el resultado del cálculo ("qué" calcular) como una simple consecuencia de las leyes de la lógica.

Nuestro lenguaje de consulta puede considerarse como un subconjunto de la lógica matemática interpretable procedimentalmente. Una afirmación representa un hecho simple (una proposición atómica). Una regla representa la implicación de que la conclusión de la regla se cumple en aquellos casos en que el cuerpo de la regla se cumple. Una regla tiene una interpretación procedimental natural: para establecer la conclusión de la regla, establezca el cuerpo de la regla. Las reglas, por lo tanto, especifican cálculos. Sin embargo, como las reglas también pueden considerarse declaraciones de lógica

matemática, podemos justificar cualquier "inferencia" lograda por un programa lógico afirmando que el mismo resultado podría obtenerse trabajando enteramente dentro de la lógica matemática.[101]

Bucles infinitos

Una consecuencia de la interpretación procedimental de los programas lógicos es que es posible construir programas irremediablemente ineficientes para resolver ciertos problemas. Un caso extremo de ineficiencia ocurre cuando el sistema cae en bucles infinitos al hacer deducciones. Como ejemplo simple, supongamos que estamos configurando una base de datos de matrimonios famosos, incluyendo

(assert! (married Minnie Mickey))

Si ahora preguntamos

(married Mickey ?who)

no obtendremos ninguna respuesta, porque el sistema no sabe que si *A* está casado con *B*, entonces *B* está casado con *A*. Así que afirmamos la regla

(assert! (rule (married ?x ?y) (married ?y ?x)))

y de nuevo consultamos

(married Mickey ?who)

Desafortunadamente, esto llevará al sistema a un bucle infinito, de la siguiente manera:

- El sistema encuentra que la regla `married` es aplicable; es decir, la conclusión de la regla `(married ?x ?y)` se unifica con éxito con el patrón de la consulta `(married Mickey ?who)` para producir un marco en el que `?x` está ligado a `Mickey` y `?y` está ligado a `?who`. Por lo tanto, el intérprete procede a evaluar el cuerpo de la regla `(married ?y ?x)` en este marco—en efecto, a procesar la consulta `(married ?who Mickey)`.
- Una respuesta aparece directamente como una aserción en la base de datos: `(married Minnie Mickey)`.
- La regla `married` también es aplicable, por lo que el intérprete vuelve a evaluar el cuerpo de la regla, que esta vez es equivalente a `(married Mickey ?who)`.

El sistema está ahora en un bucle infinito. De hecho, si el sistema encontrará la respuesta simple `(married Minnie Mickey)` antes de entrar en el bucle depende de los detalles de implementación sobre el orden en que el sistema verifica los elementos en la base de datos. Este es un ejemplo muy simple de los tipos de bucles que pueden ocurrir. Las colecciones de reglas interrelacionadas pueden llevar a bucles que son mucho más difíciles de anticipar, y la aparición de un bucle puede depender del orden de las cláusulas en un `and` (ver Ejercicio 4.64) o de detalles de bajo nivel relacionados con el orden en que el sistema procesa las consultas.[102]

Problemas con `not`

Otra peculiaridad del sistema de consulta concierne a `not`. Dada la base de datos de Section 4.4.1, considere las siguientes dos consultas:

(and (supervisor ?x ?y) (not (job ?x (computer programmer)))) (and (not (job ?x (computer programmer))) (supervisor ?x ?y))

Estas dos consultas no producen el mismo resultado. La primera consulta comienza por encontrar todas las entradas en la base de datos que coinciden con `(supervisor ?x ?y)`, y luego filtra los marcos resultantes eliminando aquellos en los que el valor de `?x` satisface `(job ?x (computer programmer))`. La segunda consulta comienza filtrando los marcos de entrada para eliminar aquellos que pueden satisfacer `(job ?x (computer programmer))`. Como el único marco de entrada está vacío, comprueba la base de datos para ver si hay algún patrón que satisfaga `(job ?x (computer programmer))`. Como generalmente hay entradas de esta forma, la cláusula `not` filtra el marco vacío y devuelve un flujo vacío de marcos. En consecuencia, toda la consulta compuesta devuelve un flujo vacío.

El problema es que nuestra implementación de `not` realmente está destinada a servir como un filtro sobre los valores de las variables. Si se procesa una cláusula `not` con un marco en el que algunas de las variables permanecen sin ligar (como lo hace `?x` en el ejemplo anterior), el sistema producirá resultados inesperados. Problemas similares ocurren con el uso de `lisp/value`—el predicado de Lisp no puede funcionar si algunos de sus argumentos no están ligados. Ver Ejercicio 4.77.

Hay también una forma mucho más seria en la que el `not` del lenguaje de consulta difiere del `not` de la lógica matemática. En lógica, interpretamos la afirmación "no *P*" como que *P* no es verdadera. En el sistema de consulta, sin embargo, "no *P*" significa que *P* no es deducible del conocimiento en la base de datos. Por ejemplo, dada la base de datos de personal de Section 4.4.1, el sistema deduciría felizmente todo tipo de enunciados `not`, tales como que

Ben Bitdiddle no es un aficionado al béisbol, que no está lloviendo afuera, y que 2 + 2 no es 4.[103] En otras palabras, el `not` de los lenguajes de programación lógica refleja la llamada *suposición de mundo cerrado* de que toda la información relevante ha sido incluida en la base de datos.[104]

Ejercicio 4.64: Louis Reasoner elimina por error la regla `outranked/by` (Section 4.4.1) de la base de datos. Cuando se da cuenta de esto, la reinstala rápidamente. Desafortunadamente, hace un ligero cambio en la regla y la escribe como

(rule (outranked-by ?staff-person ?boss) (or (supervisor ?staff-person ?boss) (and (outranked-by ?middle-manager ?boss) (supervisor ?staff-person ?middle-manager))))

Justo después de que Louis escribe esta información en el sistema, DeWitt Aull llega para averiguar quién supera en rango a Ben Bitdiddle. Emite la consulta

(outranked-by (Bitdiddle Ben) ?who)

Después de responder, el sistema entra en un bucle infinito. Explique por qué.

Ejercicio 4.65: Cy D. Fect, esperando el día en que ascenderá en la organización, realiza una consulta para encontrar a todos los peces gordos (usando la regla `wheel` de Section 4.4.1):

(wheel ?who)

Para su sorpresa, el sistema responde

;;; Query results: (wheel (Warbucks Oliver)) (wheel (Bitdiddle Ben)) (wheel (Warbucks Oliver)) (wheel (Warbucks Oliver)) (wheel (Warbucks Oliver))

¿Por qué Oliver Warbucks aparece cuatro veces?

Ejercicio 4.66: Ben ha estado generalizando el sistema de consulta para proporcionar estadísticas sobre la empresa. Por ejemplo, para encontrar el salario total de todos los programadores de computadoras se podrá decir

(sum ?amount (and (job ?x (computer programmer)) (salary ?x ?amount)))

En general, el nuevo sistema de Ben permite expresiones de la forma

(accumulation-function ⟨ *variable* ⟩ ⟨ *query pattern* ⟩)

donde `accumulation/function` puede ser cosas como `sum`, `average`, o `maximum`. Ben razona que debería ser pan comido implementar esto. Simplemente alimentará el patrón de consulta a `qeval`. Esto producirá un flujo de marcos. Luego pasará este flujo a través de una función de mapeo que extrae el valor de la variable designada de cada marco en el flujo y alimentará el flujo de valores resultante a la función de acumulación. Justo cuando Ben completa la implementación y está a punto de probarla, Cy pasa, todavía perplejo por el resultado de la consulta `wheel` en el Ejercicio 4.65. Cuando Cy le muestra a Ben la respuesta del sistema, Ben gime, "¡Oh, no, mi simple esquema de acumulación no funcionará!"

¿De qué se acaba de dar cuenta Ben? Esboce un método que pueda usar para salvar la situación.

Ejercicio 4.67: Diseñe una forma de instalar un detector de bucles en el sistema de consulta para evitar los tipos de bucles simples ilustrados en el texto y en el Ejercicio 4.64. La idea general es que el sistema debe mantener algún tipo de historial de su cadena actual de deducciones y no debe comenzar a procesar una consulta en la que ya está trabajando. Describa qué tipo de información (patrones y marcos) se incluye en este historial, y cómo debe hacerse la verificación. (Después de estudiar los detalles de la implementación del sistema de consulta en la Sección 4.4.4, es posible que desee modificar el sistema para incluir su detector de bucles.)

Ejercicio 4.68: Defina reglas para implementar la operación `reverse` del Ejercicio 2.18, que devuelve una lista que contiene los mismos elementos que una lista dada en orden inverso. (Pista: Use `append/to/form`.) ¿Pueden sus reglas responder tanto `(reverse (1 2 3) ?x)` como `(reverse ?x (1 2 3))`?

Ejercicio 4.69: Comenzando con la base de datos y las reglas que formuló en el Ejercicio 4.63, diseñe una regla para agregar "bis" a una relación de nieto. Esto debería permitir que el sistema deduzca que Irad es el bisnieto de Adán, o que Jabal y Jubal son los tataranietos de Adán. (Pista: Represente el hecho sobre Irad, por ejemplo, como `((great grandson) Adam Irad)`. Escriba reglas que determinen si una lista termina en la palabra `grandson`. Use esto para expresar una regla que permita derivar la relación `((great . ?rel) ?x ?y)`, donde `?rel` es una lista que termina en `grandson`.) Compruebe sus reglas en consultas como `((great grandson) ?g ?ggs)` y `(?relationship Adam Irad)`.

Implementando el Sistema de Consulta

Section 4.4.2 describió cómo funciona el sistema de consulta. Ahora completamos los detalles presentando una implementación completa del sistema.

El Bucle del Controlador y la Instanciación

El bucle del controlador para el sistema de consulta lee repetidamente expresiones de entrada. Si la expresión es una regla o aserción que se va a agregar a la base de datos, entonces la información se agrega. De lo contrario, se asume que la expresión es una consulta. El controlador pasa esta consulta al evaluador `qeval` junto con un flujo de marcos inicial que consiste en un solo marco vacío. El resultado de la evaluación es un flujo de marcos generado al satisfacer la consulta con los valores de las variables encontrados en la base de datos. Estos marcos se utilizan para formar un nuevo flujo que consiste en copias de la consulta original en las que las variables se instancian con los valores proporcionados por el flujo de marcos, y este flujo final se imprime en la terminal:

(define input-prompt ";;; Query input:") (define output-prompt ";;; Query results:")

(define (query-driver-loop) (prompt-for-input input-prompt) (let ((q (query-syntax-process (read)))) (cond ((assertion-to-be-added? q) (add-rule-or-assertion! (add-assertion-body q)) (newline) (display "Assertion added to data base.") (query-driver-loop)) (else (newline) (display output-prompt) (display-stream (stream-map (lambda (frame) (instantiate q frame (lambda (v f) (contract-question-mark v)))) (qeval q (singleton-stream '()))))) (query-driver-loop)))))

Aquí, como en los otros evaluadores de este capítulo, usamos una sintaxis abstracta para las expresiones del lenguaje de consulta. La implementación de la sintaxis de las expresiones, incluyendo el predicado `assertion/to/be/added?` y el selector `add/assertion/body`, se da en Section 4.4.4.7. `add/rule/or/assertion!` se define en Section 4.4.4.5.

Antes de realizar cualquier procesamiento en una expresión de entrada, el bucle del controlador la transforma sintácticamente a una forma que hace que el procesamiento sea más eficiente. Esto implica cambiar la representación de las variables de patrón. Cuando la consulta es instanciada, cualquier variable que permanezca sin ligar se transforma de nuevo a la representación de entrada antes de ser impresa. Estas transformaciones se realizan por los dos procedimientos `query/syntax/process` y `contract/`

`question/mark` (Section 4.4.4.7).

Para instanciar una expresión, la copiamos, reemplazando cualquier variable en la expresión por sus valores en un marco dado. Los valores se instancian a su vez, ya que podrían contener variables (por ejemplo, si `?x` en `exp` está ligado a `?y` como resultado de la unificación y `?y` está a su vez ligado a 5). La acción a tomar si una variable no puede ser instanciada se da mediante un argumento procedimental a `instantiate`.

(define (instantiate exp frame unbound-var-handler) (define (copy exp) (cond ((var? exp) (let ((binding (binding-in-frame exp frame))) (if binding (copy (binding-value binding)) (unbound-var-handler exp frame)))) ((pair? exp) (cons (copy (car exp)) (copy (cdr exp)))) (else exp))) (copy exp))

Los procedimientos que manipulan las ligaduras se definen en Section 4.4.4.8.

El Evaluador

El procedimiento `qeval`, llamado por el `query/driver/loop`, es el evaluador básico del sistema de consulta. Toma como entradas una consulta y un flujo de marcos, y devuelve un flujo de marcos extendidos. Identifica las formas especiales mediante un despacho dirigido por datos usando `get` y `put`, tal como hicimos al implementar operaciones genéricas en el Capítulo 2. Cualquier consulta que no se identifique como una forma especial se asume que es una consulta simple, para ser procesada por `simple/query`.

(define (qeval query frame-stream) (let ((qproc (get (type query) 'qeval))) (if qproc (qproc (contents query) frame-stream) (simple-query query frame-stream))))

`type` y `contents`, definidos en Section 4.4.4.7, implementan la sintaxis abstracta de las formas especiales.

Consultas simples

El procedimiento `simple/query` maneja las consultas simples. Toma como argumentos una consulta simple (un patrón) junto con un flujo de marcos, y devuelve el flujo formado al extender cada marco por todas las coincidencias de la base de datos de la consulta.

(define (simple-query query-pattern frame-stream) (stream-flatmap (lambda (frame) (stream-append-delayed (find-assertions query-pattern frame) (delay (apply-rules query-pattern frame)))) frame-stream))

Para cada marco en el flujo de entrada, usamos `find/assertions` (Section 4.4.4.3) para hacer coincidir el patrón con todas las aserciones en la base de datos, produciendo un flujo de marcos extendidos, y usamos `apply/rules` (Section 4.4.4.4) para aplicar todas las reglas posibles, produciendo otro flujo de marcos extendidos. Estos dos flujos se combinan (usando `stream/append/delayed`, Section 4.4.4.6) para hacer un flujo de todas las formas en que el patrón dado puede ser satisfecho de manera consistente con el marco original (ver Ejercicio 4.71). Los flujos para los marcos de entrada individuales se combinan usando `stream/flatmap` (Section 4.4.4.6) para formar un gran flujo de todas las formas en que cualquiera de los marcos en el flujo de entrada original puede ser extendido para producir una coincidencia con el patrón dado.

Consultas compuestas

Las consultas `and` se manejan como se ilustra en la Figura 4.5 por el procedimiento `conjoin`. `conjoin` toma como entradas los conjuntivos y el flujo de marcos y devuelve el flujo de marcos extendidos. Primero, `conjoin` procesa el flujo de marcos para encontrar el flujo de todas las posibles extensiones de marcos que satisfacen la primera consulta de la conjunción. Luego, usando esto como el nuevo flujo de marcos, aplica recursivamente `conjoin` al resto de las consultas.

(define (conjoin conjuncts frame-stream) (if (empty-conjunction? conjuncts) frame-stream (conjoin (rest-conjuncts conjuncts) (qeval (first-conjunct conjuncts) frame-stream))))

La expresión

(put ’and ’qeval conjoin)

configura `qeval` para despachar a `conjoin` cuando se encuentra una forma `and`.

Las consultas `or` se manejan de manera similar, como se muestra en la Figura 4.6. Los flujos de salida para las diversas disyunciones del `or` se calculan por separado y se fusionan utilizando el procedimiento `interleave/delayed` de Section 4.4.4.6. (Ver Ejercicio 4.71 y Ejercicio 4.72.)

(define (disjoin disjuncts frame-stream) (if (empty-disjunction? disjuncts) the-empty-stream (interleave-delayed (qeval (first-disjunct disjuncts) frame-stream) (delay (disjoin (rest-disjuncts disjuncts) frame-stream))))) (put ’or ’qeval disjoin)

Los predicados y selectores para la sintaxis de los conjuntivos y disyuntivos se dan en Section 4.4.4.7.

Filtros

`not` es manejado por el método descrito en Section 4.4.2. Intentamos extender cada marco en el flujo de entrada para satisfacer la consulta que se está negando, e incluimos un marco dado en el flujo de salida solo si no se puede extender.

(define (negate operands frame-stream) (stream-flatmap (lambda (frame) (if (stream-null? (qeval (negated-query operands) (singleton-stream frame))) (singleton-stream frame) the-empty-stream)) frame-stream)) (put 'not 'qeval negate)

`lisp/value` es un filtro similar a `not`. Cada marco en el flujo se usa para instanciar las variables en el patrón, se aplica el predicado indicado, y los marcos para los que el predicado devuelve falso se filtran del flujo de entrada. Se produce un error si hay variables de patrón no ligadas.

(define (lisp-value call frame-stream) (stream-flatmap (lambda (frame) (if (execute (instantiate call frame (lambda (v f) (error "Unknown pat var: LISP-VALUE" v)))) (singleton-stream frame) the-empty-stream)) frame-stream)) (put 'lisp-value 'qeval lisp-value)

`execute`, que aplica el predicado a los argumentos, debe hacer `eval` de la expresión del predicado para obtener el procedimiento a aplicar. Sin embargo, no debe evaluar los argumentos, ya que ya son los argumentos reales, no expresiones cuya evaluación (en Lisp) producirá los argumentos. Nótese que `execute` se implementa usando `eval` y `apply` del sistema Lisp subyacente.

(define (execute exp) (apply (eval (predicate exp) user-initial-environment) (args exp)))

La forma especial `always/true` proporciona una consulta que siempre se satisface. Ignora su contenido (normalmente vacío) y simplemente pasa todos los marcos del flujo de entrada. `always/true` es utilizado por el selector `rule/body` (Section 4.4.4.7) para proporcionar cuerpos para reglas que se definieron sin cuerpos (es decir, reglas cuyas conclusiones siempre se satisfacen).

(define (always-true ignore frame-stream) frame-stream) (put 'always-true 'qeval always-true)

Los selectores que definen la sintaxis de `not` y `lisp/value` se dan en Section 4.4.4.7.

Encontrando Aserciones por Coincidencia de Patrones

`find/assertions`, llamado por `simple/query` (Section 4.4.4.2), toma como entrada un patrón y un marco. Devuelve un flujo de marcos, cada uno extendiendo el dado por una coincidencia de la base de datos del patrón dado. Utiliza `fetch/assertions` (Section 4.4.4.5) para obtener un flujo de todas las aserciones en la base de datos que deben ser verificadas para una coincidencia con el patrón y el marco. La razón de `fetch/assertions` aquí es que a menudo podemos aplicar pruebas simples que eliminarán muchas de las entradas en la base de datos del grupo de candidatos para una coincidencia exitosa. El sistema seguiría funcionando si eliminásemos `fetch/assertions` y simplemente verificáramos un flujo de todas las aserciones en la base de datos, pero el cálculo sería menos eficiente porque necesitaríamos hacer muchas más llamadas al comparador.

(define (find-assertions pattern frame) (stream-flatmap (lambda (datum) (check-an-assertion datum pattern frame)) (fetch-assertions pattern frame)))

`check/an/assertion` toma como argumentos un patrón, un objeto de datos (aserción) y un marco y devuelve o un flujo de un solo elemento que contiene el marco extendido o `the/empty/stream` si la coincidencia falla.

(define (check-an-assertion assertion query-pat query-frame) (let ((match-result (pattern-match query-pat assertion query-frame))) (if (eq? match-result 'failed) the-empty-stream (singleton-stream match-result))))

El comparador de patrones básico devuelve el símbolo `failed` o una extensión del marco dado. La idea básica del comparador es verificar el patrón contra los datos, elemento por elemento, acumulando ligaduras para las variables del patrón. Si el patrón y el objeto de datos son iguales, la coincidencia tiene éxito y devolvemos el marco de ligaduras acumulado hasta ahora. De lo contrario, si el patrón es una variable, extendemos el marco actual ligando la variable a los datos, siempre que esto sea consistente con las ligaduras que ya están en el marco. Si el patrón y los datos son ambos pares, comparamos (recursivamente) el `car` del patrón con el `car` de los datos para producir un marco; en este marco luego comparamos el `cdr` del patrón con el `cdr` de los datos. Si ninguno de estos casos es aplicable, la coincidencia falla y devolvemos el símbolo `failed`.

(define (pattern-match pat dat frame) (cond ((eq? frame 'failed) 'failed)

((equal? pat dat) frame) ((var? pat) (extend-if-consistent pat dat frame)) ((and (pair? pat) (pair? dat)) (pattern-match (cdr pat) (cdr dat) (pattern-match (car pat) (car dat) frame))) (else 'failed)))

Aquí está el procedimiento que extiende un marco agregando una nueva ligadura, si esto es consistente con las ligaduras que ya están en el marco:

(define (extend-if-consistent var dat frame) (let ((binding (binding-in-frame var frame))) (if binding (pattern-match (binding-value binding) dat frame) (extend var dat frame))))

Si no hay ninguna ligadura para la variable en el marco, simplemente agregamos la ligadura de la variable a los datos. De lo contrario, hacemos coincidir, en el marco, los datos con el valor de la variable en el marco. Si el valor almacenado contiene solo constantes, como debe ser si se almacenó durante la coincidencia de patrones por `extend/if/consistent`, entonces la coincidencia simplemente prueba si los valores almacenados y los nuevos son iguales. Si es así, devuelve el marco sin modificar; si no, devuelve una indicación de fallo. Sin embargo, el valor almacenado puede contener variables de patrón si se almacenó durante la unificación (ver Section 4.4.4.4). La coincidencia recursiva del patrón almacenado con los nuevos datos agregará o verificará las ligaduras para las variables en este patrón. Por ejemplo, supongamos que tenemos un marco en el que `?x` está ligado a `(f ? y)` y `?y` no está ligado, y deseamos aumentar este marco con una ligadura de `?x` a `(f b)`. Buscamos `?x` y encontramos que está ligado a `(f ?y)`. Esto nos lleva a hacer coincidir `(f ?y)` con el nuevo valor propuesto `(f b)` en el mismo marco. Eventualmente, esta coincidencia extiende el marco agregando una ligadura de `?y` a `b`. `?X` permanece ligado a `(f ?y)`. Nunca modificamos una ligadura almacenada y nunca almacenamos más de una ligadura para una variable dada.

Los procedimientos utilizados por `extend/if/consistent` para manipular ligaduras se definen en Section 4.4.4.8.

Patrones con colas punteadas

Si un patrón contiene un punto seguido de una variable de patrón, la variable de patrón coincide con el resto de la lista de datos (en lugar del siguiente elemento de la lista de datos), tal como se esperaría con la notación de cola punteada descrita en el Ejercicio 2.20. Aunque el comparador de patrones que acabamos de implementar no busca puntos, se comporta como queremos. Esto se debe a que la primitiva `read` de Lisp, que es utilizada por `query/driver/loop` para leer la consulta y representarla como una estructura de lista, trata los puntos de una manera especial.

Cuando `read` ve un punto, en lugar de hacer que el siguiente elemento sea el siguiente elemento de una lista (el `car` de un `cons` cuyo `cdr` será el resto de la lista) hace que el siguiente elemento sea el `cdr` de la estructura de lista. Por ejemplo, la estructura de lista producida por `read` para el patrón `(computer ?type)` podría construirse evaluando la expresión `(cons 'computer (cons '?type '()))`, y la de `(computer . ?type)` podría construirse evaluando la expresión `(cons 'computer '?type)`.

Así, a medida que `pattern/match` compara recursivamente los `car`s y `cdr`s de una lista de datos y un patrón que tenía un punto, eventualmente coincide con la variable después del punto (que es un `cdr` del patrón) con una sublista de la lista de datos, ligando la variable a esa lista. Por ejemplo, hacer coincidir el patrón `(computer . ?type)` con `(computer programmer trainee)` hará coincidir `?type` con la lista `(programmer trainee)`.

Reglas y Unificación

`apply/rules` es el análogo de la regla de `find/assertions` (Sección 4.4.4.3). Toma como entrada un patrón y un marco, y forma un flujo de marcos de extensión aplicando reglas de la base de datos. `stream/flatmap` mapea `apply/a/rule` sobre el flujo de reglas posiblemente aplicables (seleccionadas por `fetch/rules`, Sección 4.4.4.5) y combina los flujos de marcos resultantes.

(define (apply-rules pattern frame) (stream-flatmap (lambda (rule) (apply-a-rule rule pattern frame)) (fetch-rules pattern frame)))

`apply/a/rule` aplica reglas usando el método descrito en Section 4.4.2. Primero aumenta su marco de argumento unificando la conclusión de la regla con el patrón en el marco dado. Si esto tiene éxito, evalúa el cuerpo de la regla en este nuevo marco.

Sin embargo, antes de que suceda nada de esto, el programa renombra todas las variables de la regla con nuevos nombres únicos. La razón de esto es evitar que las variables de diferentes aplicaciones de reglas se confundan entre sí. Por ejemplo, si dos reglas usan una variable llamada `?x`, entonces cada una puede agregar una ligadura para `?x` al marco cuando se aplica. Estos dos `?x` no tienen nada que ver entre sí, y no debemos engañarnos pensando que las dos ligaduras deben ser consistentes. En lugar de renombrar las variables, podríamos idear una estructura de entorno más inteligente; sin embargo, el enfoque de renombramiento que hemos elegido aquí es el más sencillo, aunque no el más eficiente. (Ver Ejercicio 4.79.) Aquí está el procedimiento `apply/a/rule`:

(define (apply-a-rule rule query-pattern query-frame) (let ((clean-rule (rename-variables-in rule))) (let ((unify-result (unify-match query-pattern (conclusion clean-rule) query-frame))) (if (eq? unify-result 'failed) the-empty-stream (qeval (rule-body clean-rule) (singleton-stream unify-result))))))

Los selectores `rule/body` y `conclusion` que extraen partes de una regla se definen en Section 4.4.4.7.

Generamos nombres de variables únicos asociando un identificador único (como un número) con cada aplicación de regla y combinando este identificador con los nombres de las variables originales. Por ejemplo, si el identificador de la aplicación de la regla es 7, podríamos cambiar cada `?x` de la regla a `?x/7` y cada `?y` de la regla a `?y/7`. (`make/new/variable` y `new/rule/application/id` se incluyen con los procedimientos de sintaxis en Section 4.4.4.7.)

(define (rename-variables-in rule) (let ((rule-application-id (new-rule-application-id))) (define (tree-walk exp) (cond ((var? exp) (make-new-variable exp rule-application-id)) ((pair? exp) (cons (tree-walk (car exp)) (tree-walk (cdr exp)))) (else exp))) (tree-walk rule)))

El algoritmo de unificación se implementa como un procedimiento que toma como entradas dos patrones y un marco y devuelve o bien el marco extendido o el símbolo `failed`. El unificador es como el comparador de patrones excepto que es simétrico—se permiten variables en ambos lados de la comparación. `unify/match` es básicamente lo mismo que `pattern/match`, excepto que hay código adicional (marcado como "`***`" a continuación) para manejar el caso en que el objeto del lado derecho de la comparación es una variable.

(define (unify-match p1 p2 frame) (cond ((eq? frame 'failed) 'failed) ((equal? p1 p2) frame) ((var? p1) (extend-if-possible p1 p2 frame)) ((var? p2) (extend-if-possible p2 p1 frame)) ; *** ((and (pair? p1) (pair? p2)) (unify-match (cdr p1) (cdr p2) (unify-match (car p1) (car p2) frame))) (else 'failed)))

En la unificación, al igual que en la coincidencia de patrones unilateral, queremos aceptar una extensión propuesta del marco solo si es consistente con las ligaduras existentes. El procedimiento `extend/if/possible` utilizado en la unificación es el mismo que el `extend/if/consistent` utilizado en la coincidencia de patrones, excepto por dos verificaciones especiales, marcadas como "`***`" en el programa a continuación. En el primer caso, si la variable que estamos tratando de hacer coincidir no está ligada, pero el valor con el que estamos tratando de hacerla coincidir es a su vez una variable (diferente), es necesario verificar si el valor está ligado, y si es así, hacer

coincidir su valor. Si ambas partes de la coincidencia no están ligadas, podemos ligar cualquiera a la otra.

La segunda comprobación se ocupa de los intentos de ligar una variable a un patrón que incluye esa variable. Tal situación puede ocurrir cada vez que una variable se repite en ambos patrones. Considere, por ejemplo, unificar los dos patrones `(?x ?x)` y `(?y` ⟨*expresión que involucra a ?y*⟩`)` en un marco donde tanto `?x` como `?y` no están ligados. Primero, `?x` se compara con `?y`, creando una ligadura de `?x` a `?y`. A continuación, el mismo `?x` se compara con la expresión dada que involucra a `?y`. Como `?x` ya está ligado a `?y`, esto resulta en la comparación de `?y` con la expresión. Si pensamos en el unificador como si encontrara un conjunto de valores para las variables del patrón que hacen que los patrones sean iguales, entonces estos patrones implican instrucciones para encontrar un `?y` tal que `?y` sea igual a la expresión que involucra a `?y`. No existe un método general para resolver tales ecuaciones, por lo que rechazamos tales ligaduras; estos casos son reconocidos por el predicado `depends/on?`.[105] Por otro lado, no queremos rechazar los intentos de ligar una variable a sí misma. Por ejemplo, considere unificar `(?x ?x)` y `(?y ?y)`. El segundo intento de ligar `?x` a `?y` compara `?y` (el valor almacenado de `?x`) con `?y` (el nuevo valor de `?x`). Esto se soluciona con la cláusula `equal?` de `unify/match`.

(define (extend-if-possible var val frame) (let ((binding (binding-in-frame var frame))) (cond (binding (unify-match (binding-value binding) val frame)) ((var? val) ; *** (let ((binding (binding-in-frame val frame))) (if binding (unify-match var (binding-value binding) frame) (extend var val frame)))) ((depends-on? val var frame) ; *** 'failed) (else (extend var val frame)))))

`depends/on?` es un predicado que prueba si una expresión propuesta para ser el valor de una variable de patrón depende de la variable. Esto debe hacerse en relación con el marco actual porque la expresión puede contener ocurrencias de una variable que ya tiene un valor que depende de nuestra variable de prueba. La estructura de `depends/on?` es un simple recorrido de árbol recursivo en el que sustituimos los valores de las variables cuando es necesario.

(define (depends-on? exp var frame) (define (tree-walk e) (cond ((var? e) (if (equal? var e) true (let ((b (binding-in-frame e frame))) (if b (tree-walk (binding-value b)) false)))) ((pair? e) (or (tree-walk (car e)) (tree-walk (cdr e)))) (else false))) (tree-walk exp))

Manteniendo la Base de Datos

Un problema importante al diseñar lenguajes de programación lógica es el de

organizar las cosas de manera que se examinen la menor cantidad posible de entradas irrelevantes de la base de datos al verificar un patrón dado. En nuestro sistema, además de almacenar todas las afirmaciones en un gran flujo, almacenamos todas las afirmaciones cuyos `car` son símbolos constantes en flujos separados, en una tabla indexada por el símbolo. Para recuperar una afirmación que pueda coincidir con un patrón, primero verificamos si el `car` del patrón es un símbolo constante. Si es así, devolvemos (para ser probadas usando el comparador) todas las afirmaciones almacenadas que tienen el mismo `car`. Si el `car` del patrón no es un símbolo constante, devolvemos todas las afirmaciones almacenadas. Métodos más inteligentes también podrían aprovechar la información en el marco, o intentar optimizar también el caso en que el `car` del patrón no es un símbolo constante. Evitamos incorporar nuestros criterios de indexación (usando el `car`, manejando solo el caso de los símbolos constantes) en el programa; en su lugar, llamamos a predicados y selectores que encarnan nuestros criterios.

(define THE-ASSERTIONS the-empty-stream) (define (fetch-assertions pattern frame) (if (use-index? pattern) (get-indexed-assertions pattern) (get-all-assertions))) (define (get-all-assertions) THE-ASSERTIONS) (define (get-indexed-assertions pattern) (get-stream (index-key-of pattern) 'assertion-stream))

`get/stream` busca un flujo en la tabla y devuelve un flujo vacío si no hay nada almacenado allí.

(define (get-stream key1 key2) (let ((s (get key1 key2))) (if s s the-empty-stream)))

Las reglas se almacenan de manera similar, usando el `car` de la conclusión de la regla. Sin embargo, las conclusiones de las reglas son patrones arbitrarios, por lo que se diferencian de las aserciones en que pueden contener variables. Un patrón cuyo `car` es un símbolo constante puede coincidir con reglas cuyas conclusiones comienzan con una variable, así como con reglas cuyas conclusiones tienen el mismo `car`. Por lo tanto, al buscar reglas que puedan coincidir con un patrón cuyo `car` es un símbolo constante, buscamos todas las reglas cuyas conclusiones comienzan con una variable, así como aquellas cuyas conclusiones tienen el mismo `car` que el patrón. Para este propósito, almacenamos todas las reglas cuyas conclusiones comienzan con una variable en un flujo separado en nuestra tabla, indexado por el símbolo `?`.

(define THE-RULES the-empty-stream) (define (fetch-rules pattern frame) (if (use-index? pattern) (get-indexed-rules pattern) (get-all-rules))) (define (get-all-rules) THE-RULES) (define (get-indexed-rules pattern) (stream-append (get-stream (index-key-of pattern) 'rule-stream) (get-stream '? 'rule-stream)))

`add/rule/or/assertion!` es utilizado por `query/driver/loop` para agregar aserciones y reglas a la base de datos. Cada elemento se almacena en el índice, si corresponde, y en un flujo de todas las aserciones o reglas en la base de datos.

(define (add-rule-or-assertion! assertion) (if (rule? assertion) (add-rule! assertion) (add-assertion! assertion))) (define (add-assertion! assertion) (store-assertion-in-index assertion) (let ((old-assertions THE-ASSERTIONS)) (set! THE-ASSERTIONS (cons-stream assertion old-assertions)) 'ok)) (define (add-rule! rule) (store-rule-in-index rule) (let ((old-rules THE-RULES)) (set! THE-RULES (cons-stream rule old-rules)) 'ok))

Para almacenar realmente una aserción o una regla, comprobamos si se puede indexar. Si es así, la almacenamos en el flujo apropiado.

(define (store-assertion-in-index assertion) (if (indexable? assertion) (let ((key (index-key-of assertion))) (let ((current-assertion-stream (get-stream key 'assertion-stream))) (put key 'assertion-stream (cons-stream assertion current-assertion-stream)))))) (define (store-rule-in-index rule) (let ((pattern (conclusion rule))) (if (indexable? pattern) (let ((key (index-key-of pattern))) (let ((current-rule-stream (get-stream key 'rule-stream))) (put key 'rule-stream (cons-stream rule current-rule-stream))))))

Los siguientes procedimientos definen cómo se utiliza el índice de la base de datos. Un patrón (una aserción o una conclusión de una regla) se almacenará en la tabla si comienza con una variable o un símbolo constante.

(define (indexable? pat) (or (constant-symbol? (car pat)) (var? (car pat))))

La clave bajo la cual se almacena un patrón en la tabla es `?` (si comienza con una variable) o el símbolo constante con el que comienza.

(define (index-key-of pat) (let ((key (car pat))) (if (var? key) '? key)))

El índice se usará para recuperar elementos que puedan coincidir con un patrón si el patrón comienza con un símbolo constante.

(define (use-index? pat) (constant-symbol? (car pat)))

> **Ejercicio 4.70:** ¿Cuál es el propósito de las ligaduras `let` en los procedimientos `add/assertion!` y `add/rule!`? ¿Qué habría de malo con la siguiente implementación de `add/assertion!`? Pista: Recuerde la definición del flujo infinito de unos en Section 3.5.2: `(define ones (cons/stream 1 ones))`.

(define (add-assertion! assertion) (store-assertion-in-index assertion) (set! THE-ASSERTIONS (cons-stream assertion THE-ASSERTIONS)) 'ok)

Operaciones de Flujo

El sistema de consulta utiliza algunas operaciones de flujo que no se presentaron en el Capítulo 3.

`stream/append/delayed` y `interleave/delayed` son como `stream/append` e `interleave` (Section 3.5.3), excepto que toman un argumento retardado (como el procedimiento `integral` en Section 3.5.4). Esto pospone el bucle en algunos casos (ver Ejercicio 4.71).

(define (stream-append-delayed s1 delayed-s2) (if (stream-null? s1) (force delayed-s2) (cons-stream (stream-car s1) (stream-append-delayed (stream-cdr s1) delayed-s2)))) (define (interleave-delayed s1 delayed-s2) (if (stream-null? s1) (force delayed-s2) (cons-stream (stream-car s1) (interleave-delayed (force delayed-s2) (delay (stream-cdr s1))))))

`stream/flatmap`, que se utiliza en todo el evaluador de consultas para mapear un procedimiento sobre un flujo de marcos y combinar los flujos de marcos resultantes, es el análogo de flujo del procedimiento `flatmap` introducido para las listas ordinarias en Section 2.2.3. Sin embargo, a diferencia de `flatmap` ordinario, acumulamos los flujos con un proceso de entrelazado, en lugar de simplemente añadirlos (ver Ejercicio 4.72 y Ejercicio 4.73).

(define (stream-flatmap proc s) (flatten-stream (stream-map proc s)))

(define (flatten-stream stream) (if (stream-null? stream) the-empty-stream (interleave-delayed (stream-car stream) (delay (flatten-stream (stream-cdr stream))))))

El evaluador también utiliza el siguiente procedimiento simple para generar un flujo que consiste en un solo elemento:

(define (singleton-stream x) (cons-stream x the-empty-stream))

Procedimientos de Sintaxis de Consulta

`type` y `contents`, utilizados por `qeval` (Section 4.4.4.2), especifican que una forma especial se identifica por el símbolo en su `car`. Son los mismos que los procedimientos `type/tag` y `contents` en Section 2.4.2, excepto por el mensaje de error.

(define (type exp) (if (pair? exp) (car exp) (error "Unknown expression TYPE" exp))) (define (contents exp) (if (pair? exp) (cdr exp) (error "Unknown expression CONTENTS" exp)))

Los siguientes procedimientos, utilizados por `query/driver/loop` (en Section 4.4.4.1), especifican que las reglas y aserciones se agregan a la base de datos mediante expresiones de la forma `(assert!` ⟨*regla/o/aserción*⟩`)`:

(define (assertion-to-be-added? exp) (eq? (type exp) 'assert!)) (define (add-assertion-body exp) (car (contents exp)))

Aquí están las definiciones de sintaxis para las formas especiales `and`, `or`, `not` y `lisp/value` (Section 4.4.4.2):

(define (empty-conjunction? exps) (null? exps)) (define (first-conjunct exps) (car exps)) (define (rest-conjuncts exps) (cdr exps)) (define (empty-disjunction? exps) (null? exps)) (define (first-disjunct exps) (car exps)) (define (rest-disjuncts exps) (cdr exps)) (define (negated-query exps) (car exps)) (define (predicate exps) (car exps)) (define (args exps) (cdr exps))

Los siguientes tres procedimientos definen la sintaxis de las reglas:

(define (rule? statement) (tagged-list? statement 'rule)) (define (conclusion rule) (cadr rule)) (define (rule-body rule) (if (null? (cddr rule)) '(always-true) (caddr rule)))

`query/driver/loop` (Section 4.4.4.1) llama a `query/syntax/process` para transformar las variables de patrón en la expresión, que tienen la forma `? símbolo`, al formato interno `(? símbolo)`. Es decir, un patrón como `(job ?x ?y)` es realmente representado internamente por el sistema como `(job (? x) (? y))`. Esto aumenta la eficiencia del procesamiento de consultas, ya que significa que el sistema puede verificar si una expresión es una variable de patrón comprobando si el `car` de la expresión es el símbolo `?`, en lugar de tener que extraer caracteres del símbolo. La transformación de sintaxis se logra mediante el siguiente procedimiento:[106]

(define (query-syntax-process exp) (map-over-symbols expand-question-mark exp)) (define (map-over-symbols proc exp) (cond ((pair? exp) (cons (map-over-symbols proc (car exp)) (map-over-symbols proc (cdr exp)))) ((symbol? exp) (proc exp)) (else exp))) (define (expand-question-mark symbol) (let ((chars (symbol->string symbol))) (if (string=? (substring chars 0 1) "?") (list '? (string->symbol (substring chars 1 (string-length chars)))) symbol)))

Una vez que las variables se transforman de esta manera, las variables en un patrón son listas que comienzan con `?`, y los símbolos constantes (que deben

ser reconocidos para la indexación de la base de datos, Section 4.4.4.5) son solo los símbolos.

(define (var? exp) (tagged-list? exp '?)) (define (constant-symbol? exp) (symbol? exp))

Las variables únicas se construyen durante la aplicación de la regla (en la Sección 4.4.4.4) por medio de los siguientes procedimientos. El identificador único para una aplicación de regla es un número, que se incrementa cada vez que se aplica una regla.

(define rule-counter 0) (define (new-rule-application-id) (set! rule-counter (+ 1 rule-counter)) rule-counter) (define (make-new-variable var rule-application-id) (cons '? (cons rule-application-id (cdr var))))

Cuando `query/driver/loop` instancia la consulta para imprimir la respuesta, convierte cualquier variable de patrón no ligada de nuevo a la forma correcta para la impresión, usando

(define (contract-question-mark variable) (string->symbol (string-append "?" (if (number? (cadr variable)) (string-append (symbol->string (caddr variable)) "-" (number->string (cadr variable))) (symbol->string (cadr variable))))))

Marcos y Ligaduras

Los marcos se representan como listas de ligaduras, que son pares variable-valor:

(define (make-binding variable value) (cons variable value)) (define (binding-variable binding) (car binding)) (define (binding-value binding) (cdr binding)) (define (binding-in-frame variable frame) (assoc variable frame)) (define (extend variable value frame) (cons (make-binding variable value) frame))

> **Ejercicio 4.71:** Louis Reasoner se pregunta por qué los procedimientos `simple/query` y `disjoin` (Section 4.4.4.2) se implementan usando operaciones `delay` explícitas, en lugar de definirse de la siguiente manera:
>
> (define (simple-query query-pattern frame-stream) (stream-flatmap (lambda (frame) (stream-append (find-assertions query-pattern frame) (apply-rules query-pattern frame))) frame-stream)) (define (disjoin disjuncts frame-stream) (if (empty-disjunction? disjuncts) the-empty-stream (interleave (qeval (first-disjunct disjuncts) frame-stream) (disjoin

(rest-disjuncts disjuncts) frame-stream))))

¿Puedes dar ejemplos de consultas en las que estas definiciones más simples llevarían a un comportamiento indeseable?

Ejercicio 4.72: ¿Por qué `disjoin` y `stream/flatmap` entrelazan los flujos en lugar de simplemente añadirlos? Dé ejemplos que ilustren por qué el entrelazado funciona mejor. (Pista: ¿Por qué usamos `interleave` en Section 3.5.3?)

Ejercicio 4.73: ¿Por qué `flatten/stream` usa `delay` explícitamente? ¿Qué habría de malo en definirlo de la siguiente manera?

(define (flatten-stream stream) (if (stream-null? stream) the-empty-stream (interleave (stream-car stream) (flatten-stream (stream-cdr stream)))))

Ejercicio 4.74: Alyssa P. Hacker propone usar una versión más simple de `stream/flatmap` en `negate`, `lisp/value`, y `find/assertions`. Ella observa que el procedimiento que se mapea sobre el flujo de marcos en estos casos siempre produce o el flujo vacío o un flujo de un solo elemento, por lo que no se necesita ningún entrelazado al combinar estos flujos.

1. Completa las expresiones que faltan en el programa de Alyssa.

   (define (simple-stream-flatmap proc s) (simple-flatten (stream-map proc s))) (define (simple-flatten stream) (stream-map ⟨ ?? ⟩ (stream-filter ⟨ ?? ⟩ stream)))

2. ¿Cambia el comportamiento del sistema de consulta si lo cambiamos de esta manera?

Ejercicio 4.75: Implemente para el lenguaje de consulta una nueva forma especial llamada `unique`. `unique` debe tener éxito si hay precisamente un elemento en la base de datos que satisface una consulta especificada. Por ejemplo,

(unique (job ?x (computer wizard)))

debería imprimir el flujo de un solo elemento

(unique (job (Bitdiddle Ben) (computer wizard)))

dado que Ben es el único mago de la computación, y

(unique (job ?x (computer programmer)))

debería imprimir el flujo vacío, ya que hay más de un programador de computadoras. Además,

(and (job ?x ?j) (unique (job ?anyone ?j)))

debería listar todos los trabajos que son ocupados por una sola persona, y las personas que los ocupan.

Hay dos partes para implementar `unique`. La primera es escribir un procedimiento que maneje esta forma especial, y la segunda es hacer que `qeval` despache a ese procedimiento. La segunda parte es trivial, ya que `qeval` hace su despacho de una manera dirigida por datos. Si su procedimiento se llama `uniquely/asserted`, todo lo que necesita hacer es

(put 'unique 'qeval uniquely-asserted)

y `qeval` despachará a este procedimiento para cada consulta cuyo `type` (`car`) sea el símbolo `unique`.

El problema real es escribir el procedimiento `uniquely/asserted`. Este debería tomar como entrada los `contents` (`cdr`) de la consulta `unique`, junto con un flujo de marcos. Para cada marco en el flujo, debería usar `qeval` para encontrar el flujo de todas las extensiones al marco que satisfacen la consulta dada. Cualquier flujo que no tenga exactamente un elemento en él debería eliminarse. Los flujos restantes deberían devolverse para ser acumulados en un gran flujo que es el resultado de la consulta `unique`. Esto es similar a la implementación de la forma especial `not`.

Pruebe su implementación formando una consulta que liste a todas las personas que supervisan precisamente a una persona.

Ejercicio 4.76: Nuestra implementación de `y` como una combinación en serie de consultas (Figura 4.5) es elegante, pero es ineficiente porque al procesar la segunda consulta del `y` debemos escanear la base de datos por cada marco producido por la primera consulta. Si la base de datos tiene n elementos, y una consulta típica produce un número de marcos de salida proporcional a n (digamos n/k), entonces escanear la base de datos por cada marco producido por la primera consulta requerirá n^2/k llamadas al comparador de patrones. Otro enfoque sería procesar las dos cláusulas del `y` por separado, y luego buscar todos los pares de marcos de salida que sean compatibles. Si cada consulta produce n/k marcos de salida, entonces esto significa que debemos realizar n^2/k^2 comprobaciones de compatibilidad—un factor de k menos que el número de coincidencias

requeridas en nuestro método actual.

Diseñe una implementación de `y` que utilice esta estrategia. Debe implementar un procedimiento que tome dos marcos como entradas, compruebe si las ligaduras en los marcos son compatibles y, si es así, produzca un marco que fusione los dos conjuntos de ligaduras. Esta operación es similar a la unificación.

Ejercicio 4.77: En Section 4.4.3 vimos que `not` y `lisp/value` pueden hacer que el lenguaje de consulta dé respuestas "incorrectas" si estas operaciones de filtrado se aplican a marcos en los que las variables no están ligadas. Diseñe una forma de corregir esta deficiencia. Una idea es realizar el filtrado de una manera "retrasada" añadiendo al marco una "promesa" de filtrar que se cumple solo cuando se han ligado suficientes variables para hacer posible la operación. Podríamos esperar a realizar el filtrado hasta que se hayan realizado todas las demás operaciones. Sin embargo, por razones de eficiencia, nos gustaría realizar el filtrado lo antes posible para reducir el número de marcos intermedios generados.

Ejercicio 4.78: Rediseñe el lenguaje de consulta como un programa no determinista para ser implementado usando el evaluador de Section 4.3, en lugar de como un proceso de flujo. En este enfoque, cada consulta producirá una única respuesta (en lugar del flujo de todas las respuestas) y el usuario puede escribir `try/again` para ver más respuestas. Debería encontrar que gran parte del mecanismo que construimos en esta sección está subsumido por la búsqueda no determinista y el retroceso. Sin embargo, probablemente también encontrará que su nuevo lenguaje de consulta tiene sutiles diferencias de comportamiento con el implementado aquí. ¿Puede encontrar ejemplos que ilustren esta diferencia?

Ejercicio 4.79: Cuando implementamos el evaluador de Lisp en Section 4.1, vimos cómo usar entornos locales para evitar conflictos de nombres entre los parámetros de los procedimientos. Por ejemplo, al evaluar

(define (square x) (* x x)) (define (sum-of-squares x y) (+ (square x) (square y))) (sum-of-squares 3 4)

no hay confusión entre la `x` en `square` y la `x` en `sum/of/squares`, porque evaluamos el cuerpo de cada procedimiento en un entorno que está especialmente construido para contener las ligaduras de las variables locales. En el sistema de consulta, usamos una estrategia diferente para evitar conflictos de nombres al aplicar reglas. Cada vez que aplicamos una regla, renombramos las variables con nuevos nombres que se

garantiza que son únicos. La estrategia análoga para el evaluador de Lisp sería eliminar los entornos locales y simplemente renombrar las variables en el cuerpo de un procedimiento cada vez que aplicamos el procedimiento.

Implemente para el lenguaje de consulta un método de aplicación de reglas que utilice entornos en lugar de renombrar. Vea si puede construir sobre su estructura de entorno para crear construcciones en el lenguaje de consulta para tratar con sistemas grandes, como el análogo de las reglas de los procedimientos estructurados en bloques. ¿Puede relacionar algo de esto con el problema de hacer deducciones en un contexto (por ejemplo, "Si supusiera que *P* fuera verdadero, entonces podría deducir *A* y *B*.") como método de resolución de problemas? (Este problema es abierto. Una buena respuesta probablemente valga un doctorado.)

1. La misma idea es omnipresente en toda la ingeniería. Por ejemplo, los ingenieros eléctricos usan muchos lenguajes diferentes para describir circuitos. Dos de estos son el lenguaje de las *redes* eléctricas y el lenguaje de los *sistemas* eléctricos. El lenguaje de redes enfatiza el modelado físico de dispositivos en términos de elementos eléctricos discretos. Los objetos primitivos del lenguaje de redes son componentes eléctricos primitivos como resistores, capacitores, inductores y transistores, que se caracterizan en términos de variables físicas llamadas voltaje y corriente. Al describir circuitos en el lenguaje de redes, el ingeniero se preocupa por las características físicas de un diseño. En contraste, los objetos primitivos del lenguaje de sistemas son módulos de procesamiento de señales como filtros y amplificadores. Solo el comportamiento funcional de los módulos es relevante, y las señales se manipulan sin preocuparse por su realización física como voltajes y corrientes. El lenguaje de sistemas se erige sobre el lenguaje de redes, en el sentido de que los elementos de los sistemas de procesamiento de señales se construyen a partir de redes eléctricas. Aquí, sin embargo, las preocupaciones son con la organización a gran escala de los dispositivos eléctricos para resolver un problema de aplicación dado; la viabilidad física de las partes se asume. Esta colección de lenguajes en capas es otro ejemplo de la técnica de diseño estratificado ilustrada por el lenguaje de imágenes de Section 2.2.4.

2. Las características más importantes que nuestro evaluador omite son los mecanismos para manejar errores y soportar la depuración. Para una discusión más extensa de los evaluadores, ver Friedman et al. 1992, que da una exposición de los lenguajes de programación que procede a través de una secuencia de evaluadores escritos en Scheme.

3. Aun así, quedarán aspectos importantes del proceso de evaluación que no son elucidados por nuestro evaluador. Los más importantes de estos son los mecanismos detallados por los cuales los procedimientos llaman a otros procedimientos y devuelven valores a sus llamadores. Abordaremos estos problemas en el Capítulo 5, donde echaremos un vistazo más de cerca al proceso de evaluación implementando el evaluador como una simple máquina de registros.

4. Si nos concedemos la capacidad de aplicar primitivas, ¿qué nos queda por implementar en el evaluador? El trabajo del evaluador no es especificar las primitivas del lenguaje, sino proporcionar el tejido conectivo—los medios de combinación y los medios de abstracción—que une una colección de primitivas para formar un lenguaje. Específicamente:

• El evaluador nos permite tratar con expresiones anidadas. Por ejemplo, aunque simplemente aplicar primitivas sería suficiente para evaluar la expresión `(+ 1 6)`, no es adecuado para manejar `(+ 1 (* 2 3))`. En lo que respecta al procedimiento primitivo `+`, sus argumentos deben ser números, y se ahogaría si le pasáramos la expresión `(* 2 3)` como argumento. Un papel importante del evaluador es coreografiar la composición de procedimientos para que `(* 2 3)` se reduzca a 6 antes de pasarlo como argumento a `+`.

• El evaluador nos permite usar variables. Por ejemplo, el procedimiento primitivo para la suma no tiene forma de tratar con expresiones como `(+ x 1)`. Necesitamos un evaluador para llevar un registro de las variables y obtener sus valores antes de invocar los procedimientos primitivos.

• El evaluador nos permite definir procedimientos compuestos. Esto implica llevar un registro de las definiciones de procedimientos, saber cómo usar estas definiciones al evaluar expresiones y proporcionar un mecanismo que permita a los procedimientos aceptar argumentos.

• El evaluador proporciona las formas especiales, que deben evaluarse de manera diferente a las llamadas a procedimientos.

5. Podríamos haber simplificado la cláusula `application?` en `eval` usando `map` (y estipulando que `operands` devuelve una lista) en lugar de escribir un procedimiento explícito `list/of/values`. Elegimos no usar `map` aquí para enfatizar el hecho de que el evaluador puede implementarse sin ningún uso de procedimientos de orden superior (y por lo tanto podría escribirse en un lenguaje que no tiene procedimientos de orden superior), incluso aunque el lenguaje que soporta incluirá procedimientos de orden superior.

6. En este caso, el lenguaje que se está implementando y el lenguaje de implementación son el mismo. La contemplación del significado de `true?`

aquí produce la expansión de la conciencia sin el abuso de sustancias.

7. Esta implementación de `define` ignora un problema sutil en el manejo de las definiciones internas, aunque funciona correctamente en la mayoría de los casos. Veremos cuál es el problema y cómo resolverlo en Section 4.1.6.

8. Como dijimos cuando introdujimos `define` y `set!`, estos valores dependen de la implementación en Scheme—es decir, el implementador puede elegir qué valor devolver.

9. Como se mencionó en Section 2.3.1, el evaluador ve una expresión citada como una lista que comienza con `quote`, incluso si la expresión se escribe con la marca de comillas. Por ejemplo, la expresión `'a` sería vista por el evaluador como `(quote a)`. Ver Ejercicio 2.55.

10. El valor de una expresión `if` cuando el predicado es falso y no hay alternativa no está especificado en Scheme; hemos elegido aquí que sea falso. Apoyaremos el uso de las variables `true` y `false` en las expresiones que se evaluarán ligándolas en el entorno global. Ver Section 4.1.4.

11. Estos selectores para una lista de expresiones—y los correspondientes para una lista de operandos—no pretenden ser una abstracción de datos. Se introducen como nombres mnemotécnicos para las operaciones básicas de lista para facilitar la comprensión del evaluador de control explícito en Section 5.4.

12. El valor de una expresión `cond` cuando todos los predicados son falsos y no hay una cláusula `else` no está especificado en Scheme; hemos elegido aquí que sea falso.

13. Los sistemas Lisp prácticos proporcionan un mecanismo que permite al usuario agregar nuevas expresiones derivadas y especificar su implementación como transformaciones sintácticas sin modificar el evaluador. Tal transformación definida por el usuario se llama una *macro*. Aunque es fácil agregar un mecanismo elemental para definir macros, el lenguaje resultante tiene sutiles problemas de conflicto de nombres. Ha habido mucha investigación sobre mecanismos para la definición de macros que no causan estas dificultades. Ver, por ejemplo, Kohlbecker 1986, Clinger y Rees 1991, y Hanson 1991.

14. Los marcos no son realmente una abstracción de datos en el siguiente código: `set/variable/value!` y `define/variable!` usan `set/car!` para modificar directamente los valores en un marco. El propósito de los procedimientos de marco es hacer que los procedimientos de manipulación de entornos sean fáciles de leer.

15. El inconveniente de esta representación (así como la variante del Ejercicio 4.11) es que el evaluador puede tener que buscar en muchos marcos para encontrar la ligadura de una variable dada. (Tal enfoque se conoce como *ligadura profunda*.) Una forma de evitar esta ineficiencia es hacer uso de una estrategia llamada *direccionamiento léxico*, que se discutirá en Section 5.5.6.

16. Cualquier procedimiento definido en el Lisp subyacente puede usarse como una primitiva para el evaluador metacircular. El nombre de una primitiva instalada en el evaluador no tiene por qué ser el mismo que el nombre de su implementación en el Lisp subyacente; los nombres son los mismos aquí porque el evaluador metacircular implementa el propio Scheme. Así, por ejemplo, podríamos poner `(list 'first car)` o `(list 'square (lambda (x) (* x x)))` en la lista de `primitive/procedures`.

17. `apply/in/underlying/scheme` es el procedimiento `apply` que hemos usado en capítulos anteriores. El procedimiento `apply` del evaluador metacircular (Section 4.1.1) modela el funcionamiento de esta primitiva. Tener dos cosas diferentes llamadas `apply` conduce a un problema técnico al ejecutar el evaluador metacircular, porque definir el `apply` del evaluador metacircular enmascarará la definición de la primitiva. Una forma de evitar esto es renombrar el `apply` metacircular para evitar conflictos con el nombre del procedimiento primitivo. Hemos asumido en cambio que hemos guardado una referencia al `apply` subyacente haciendo

(define apply-in-underlying-scheme apply)

antes de definir el `apply` metacircular. Esto nos permite acceder a la versión original de `apply` con un nombre diferente.

18. El procedimiento primitivo `read` espera la entrada del usuario y devuelve la siguiente expresión completa que se escribe. Por ejemplo, si el usuario escribe `(+ 23 x)`, `read` devuelve una lista de tres elementos que contiene el símbolo `+`, el número 23 y el símbolo `x`. Si el usuario escribe `'x`, `read` devuelve una lista de dos elementos que contiene el símbolo `quote` y el símbolo `x`.

19. El hecho de que las máquinas se describan en Lisp es inesencial. Si le damos a nuestro evaluador un programa de Lisp que se comporta como un evaluador para algún otro lenguaje, digamos C, el evaluador de Lisp emulará el evaluador de C, que a su vez puede emular cualquier máquina descrita como un programa de C. De manera similar, escribir un evaluador de Lisp en C produce un programa de C que puede ejecutar cualquier programa de Lisp. La idea profunda aquí es que cualquier evaluador puede emular a cualquier

otro. Por lo tanto, la noción de "lo que en principio puede ser calculado" (ignorando las cuestiones prácticas de tiempo y memoria requeridos) es independiente del lenguaje o de la computadora, y en cambio refleja una noción subyacente de *computabilidad*. Esto fue demostrado por primera vez de una manera clara por Alan M. Turing (1912-1954), cuyo artículo de 1936 sentó las bases de la informática teórica. En el artículo, Turing presentó un modelo computacional simple, ahora conocido como una *máquina de Turing*, y argumentó que cualquier "proceso efectivo" puede formularse como un programa para tal máquina. (Este argumento se conoce como la *tesis de Church-Turing*.) Turing luego implementó una máquina universal, es decir, una máquina de Turing que se comporta como un evaluador para los programas de la máquina de Turing. Utilizó este marco para demostrar que hay problemas bien planteados que no pueden ser calculados por las máquinas de Turing (ver Ejercicio 4.15), y por lo tanto, por implicación, no pueden formularse como "procesos efectivos". Turing pasó a hacer contribuciones fundamentales a la informática práctica también. Por ejemplo, inventó la idea de estructurar programas usando subrutinas de propósito general. Ver Hodges 1983 para una biografía de Turing.

20. Algunas personas encuentran contraintuitivo que un evaluador, que se implementa mediante un procedimiento relativamente simple, pueda emular programas que son más complejos que el propio evaluador. La existencia de una máquina evaluadora universal es una propiedad profunda y maravillosa de la computación. La *teoría de la recursión*, una rama de la lógica matemática, se ocupa de los límites lógicos de la computación. El hermoso libro de Douglas Hofstadter, *Gödel, Escher, Bach*, explora algunas de estas ideas (Hofstadter 1979).

21. Advertencia: Esta primitiva `eval` no es idéntica al procedimiento `eval` que implementamos en la Sección 4.1.1, porque utiliza entornos de Scheme *reales* en lugar de las estructuras de entorno de ejemplo que construimos en la Sección 4.1.3. Estos entornos reales no pueden ser manipulados por el usuario como listas ordinarias; deben ser accedidos a través de `eval` u otras operaciones especiales. Del mismo modo, la primitiva `apply` que vimos anteriormente no es idéntica al `apply` metacircular, porque utiliza procedimientos reales de Scheme en lugar de los objetos de procedimiento que construimos en la Sección 4.1.3 y la Sección 4.1.4.

22. La implementación de Scheme de MIT incluye `eval`, así como un símbolo `user/initial/environment` que está ligado al entorno inicial en el que se evalúan las expresiones de entrada del usuario.

23. Aunque estipulamos que a `halts?` se le da un objeto de procedimiento, nótese que este razonamiento todavía se aplica incluso si `halts?` puede

acceder al texto del procedimiento y a su entorno. Este es el célebre *Teorema de la Parada* de Turing, que dio el primer ejemplo claro de un problema *no computable*, es decir, una tarea bien planteada que no puede llevarse a cabo como un procedimiento computacional.

24. Querer que los programas no dependan de este mecanismo de evaluación es la razón de la observación de que "la gerencia no es responsable" en la Nota 28 del Capítulo 1. Al insistir en que las definiciones internas vengan primero y no se usen entre sí mientras se evalúan las definiciones, el estándar IEEE para Scheme deja a los implementadores cierta elección en el mecanismo utilizado para evaluar estas definiciones. La elección de una regla de evaluación en lugar de otra aquí puede parecer un problema menor, que afecta solo a la interpretación de programas "mal formados". Sin embargo, veremos en Section 5.5.6 que pasar a un modelo de ámbito simultáneo para las definiciones internas evita algunas dificultades desagradables que de otro modo surgirían al implementar un compilador.

25. El estándar IEEE para Scheme permite diferentes estrategias de implementación al especificar que es responsabilidad del programador obedecer esta restricción, no de la implementación forzarla. Algunas implementaciones de Scheme, incluida la de MIT Scheme, utilizan la transformación mostrada anteriormente. Por lo tanto, algunos programas que no obedecen esta restricción se ejecutarán de hecho en tales implementaciones.

26. Los implementadores de Scheme de MIT apoyan a Alyssa por las siguientes razones: Eva tiene en principio la razón—las definiciones deberían considerarse simultáneas. Pero parece difícil implementar un mecanismo general y eficiente que haga lo que Eva requiere. En ausencia de tal mecanismo, es mejor generar un error en los casos difíciles de definiciones simultáneas (la noción de Alyssa) que producir una respuesta incorrecta (como lo haría Ben).

27. Este ejemplo ilustra un truco de programación para formular procedimientos recursivos sin usar `define`. El truco más general de este tipo es el *operador Y*, que se puede usar para dar una implementación de "cálculo lambda puro" de la recursión. (Ver Stoy 1977 para detalles sobre el cálculo lambda, y Gabriel 1988 para una exposición del operador *Y* en Scheme.)

28. Esta técnica es una parte integral del proceso de compilación, que discutiremos en el Capítulo 5. Jonathan Rees escribió un intérprete de Scheme como este alrededor de 1982 para el proyecto T (Rees and Adams 1982). Marc Feeley (1986) (ver también Feeley and Lapalme 1987) inventó independientemente esta técnica en su tesis de maestría.

29. Sin embargo, hay una parte importante de la búsqueda de variables que *puede* hacerse como parte del análisis sintáctico. Como mostraremos en Section 5.5.6, se puede determinar la posición en la estructura del entorno donde se encontrará el valor de la variable, obviando así la necesidad de escanear el entorno para la entrada que coincide con la variable.

30. Ver Ejercicio 4.23 para una idea del procesamiento de secuencias.

31. Snarf: "Agarrar, especialmente un documento o archivo grande con el propósito de usarlo con o sin el permiso del propietario". Snarf down: "Hacer snarf, a veces con la connotación de absorber, procesar o comprender." (Estas definiciones fueron tomadas prestadas de Steele et al. 1983. Ver también Raymond 1993.)

32. La diferencia entre la terminología "perezosa" y la terminología "de orden normal" es algo difusa. Generalmente, "perezoso" se refiere a los mecanismos de evaluadores particulares, mientras que "de orden normal" se refiere a la semántica de los lenguajes, independientemente de cualquier estrategia de evaluación particular. Pero esta no es una distinción rígida, y las dos terminologías a menudo se usan indistintamente.

33. La terminología "estricto" versus "no estricto" significa esencialmente lo mismo que "orden aplicativo" versus "orden normal", excepto que se refiere a procedimientos y argumentos individuales en lugar de al lenguaje en su totalidad. En una conferencia sobre lenguajes de programación, podría escuchar a alguien decir: "El lenguaje de orden normal Hassle tiene ciertas primitivas estrictas. Otros procedimientos toman sus “argumentos por evaluación perezosa”

34. La palabra *thunk* fue inventada por un grupo de trabajo informal que estaba discutiendo la implementación de la llamada por nombre en Algol 60. Observaron que la mayor parte del análisis ("pensar en") la expresión podía hacerse en tiempo de compilación; así, en tiempo de ejecución, la expresión ya habría sido "pensada" (Ingerman et al. 1960).

35. Esto es análogo al uso de `force` en los objetos retardados que se introdujeron en el Capítulo 3 para representar flujos. La diferencia crítica entre lo que estamos haciendo aquí y lo que hicimos en el Capítulo 3 es que estamos incorporando el retraso y el forzamiento en el evaluador, y por lo tanto haciendo esto uniforme y automático en todo el lenguaje.

36. La evaluación perezosa combinada con la memoización a veces se denomina paso de argumentos por *llamada por necesidad*, en contraste con el paso de argumentos por *llamada por nombre*. (La llamada por nombre, introducida en Algol 60, es similar a la evaluación perezosa no memoizada.)

Como diseñadores de lenguajes, podemos construir nuestro evaluador para memoizar, no para memoizar, o dejar esto como una opción para los programadores (Ejercicio 4.31). Como podría esperarse del Capítulo 3, estas opciones plantean problemas que se vuelven sutiles y confusos en presencia de asignaciones. (Ver Ejercicio 4.27 y Ejercicio 4.29.) Un excelente artículo de Clinger (1982) intenta aclarar las múltiples dimensiones de confusión que surgen aquí.

37. Nótese que también borramos el `env` del thunk una vez que se ha calculado el valor de la expresión. Esto no supone ninguna diferencia en los valores devueltos por el intérprete. Sin embargo, ayuda a ahorrar espacio, porque eliminar la referencia del thunk al `env` una vez que ya no es necesario permite que esta estructura sea *recolectada como basura* y su espacio reciclado, como discutiremos en Section 5.3. De manera similar, podríamos haber permitido que los entornos innecesarios en los objetos retardados memoizados de Section 3.5.1 fueran recolectados como basura, haciendo que `memo/proc` hiciera algo como `(set! proc '())` para descartar el procedimiento `proc` (que incluye el entorno en el que se evaluó el `delay`) después de almacenar su valor.

38. Este ejercicio demuestra que la interacción entre la evaluación perezosa y los efectos secundarios puede ser muy confusa. Esto es justo lo que se podría esperar de la discusión en el Capítulo 3.

39. Este es precisamente el problema con el procedimiento `unless`, como en el Ejercicio 4.26.

40. Esta es la representación procedimental descrita en el Ejercicio 2.4. Esencialmente, cualquier representación procedimental (por ejemplo, una implementación de paso de mensajes) funcionaría igual de bien. Nótese que podemos instalar estas definiciones en el evaluador perezoso simplemente escribiéndolas en el bucle del controlador. Si originalmente hubiéramos incluido `cons`, `car` y `cdr` como primitivas en el entorno global, se redefinirán. (También ver Ejercicio 4.33 y Ejercicio 4.34.)

41. Esto nos permite crear versiones retardadas de tipos más generales de estructuras de lista, no solo secuencias. Hughes 1990 discute algunas aplicaciones de los "árboles perezosos".

42. Asumimos que hemos definido previamente un procedimiento `prime?` que prueba si los números son primos. Incluso con `prime?` definido, el procedimiento `prime/sum/pair` puede parecer sospechosamente similar al inútil intento de "pseudo-Lisp" de definir la función de raíz cuadrada, que describimos al principio de Section 1.1.7. De hecho, un procedimiento de raíz

cuadrada en esa línea puede formularse realmente como un programa no determinista. Al incorporar un mecanismo de búsqueda en el evaluador, estamos erosionando la distinción entre descripciones puramente declarativas y especificaciones imperativas de cómo calcular las respuestas. Iremos aún más lejos en esta dirección en Section 4.4.

43. La idea de `amb` para la programación no determinista fue descrita por primera vez en 1961 por John McCarthy (ver McCarthy 1963).

44. En realidad, la distinción entre devolver no determinísticamente una sola elección y devolver todas las elecciones depende en cierto modo de nuestro punto de vista. Desde la perspectiva del código que utiliza el valor, la elección no determinista devuelve un solo valor. Desde la perspectiva del programador que diseña el código, la elección no determinista devuelve potencialmente todos los valores posibles, y el cálculo se ramifica de modo que cada valor se investiga por separado.

45. Se podría objetar que este es un mecanismo irremediablemente ineficiente. Podría requerir millones de procesadores para resolver algún problema fácilmente planteado de esta manera, y la mayor parte del tiempo la mayoría de esos procesadores estarían inactivos. Esta objeción debe tomarse en el contexto de la historia. La memoria solía considerarse un producto caro. En 1964 un megabyte de RAM costaba alrededor de $400,000. Ahora cada computadora personal tiene muchos megabytes de RAM, y la mayor parte del tiempo la mayor parte de esa RAM no se utiliza. Es difícil subestimar el costo de la electrónica producida en masa.

46. Automágicamente: "Automáticamente, pero de una manera que, por alguna razón (típicamente porque es demasiado complicado, o demasiado feo, o quizás incluso demasiado trivial), el hablante no tiene ganas de explicar". (Steele et al. 1983, Raymond 1993)

47. La integración de estrategias de búsqueda automática en los lenguajes de programación ha tenido una historia larga y accidentada. Las primeras sugerencias de que los algoritmos no deterministas podrían codificarse elegantemente en un lenguaje de programación con búsqueda y retroceso automático provinieron de Robert Floyd (1967). Carl Hewitt (1969) inventó un lenguaje de programación llamado Planner que soportaba explícitamente el retroceso cronológico automático, proporcionando una estrategia de búsqueda en profundidad incorporada. Sussman et al. (1971) implementaron un subconjunto de este lenguaje, llamado MicroPlanner, que se utilizó para apoyar el trabajo en la resolución de problemas y la planificación de robots. Ideas similares, surgidas de la lógica y la demostración de teoremas, llevaron a la génesis en Edimburgo y Marsella del elegante lenguaje Prolog (que

discutiremos en Section 4.4). Después de una frustración suficiente con la búsqueda automática, McDermott y Sussman (1972) desarrollaron un lenguaje llamado Conniver, que incluía mecanismos para poner la estrategia de búsqueda bajo el control del programador. Sin embargo, esto resultó engorroso, y Sussman y Stallman 1975 encontraron un enfoque más manejable mientras investigaban métodos de análisis simbólico para circuitos eléctricos. Desarrollaron un esquema de retroceso no cronológico que se basaba en rastrear las dependencias lógicas que conectan los hechos, una técnica que ha llegado a conocerse como *retroceso dirigido por dependencias*. Aunque su método era complejo, produjo programas razonablemente eficientes porque realizaba poca búsqueda redundante. Doyle (1979) y McAllester (1978; 1980) generalizaron y clarificaron los métodos de Stallman y Sussman, desarrollando un nuevo paradigma para formular la búsqueda que ahora se llama *mantenimiento de la verdad*. Todos los sistemas modernos de resolución de problemas utilizan alguna forma de sistema de mantenimiento de la verdad como sustrato. Ver Forbus y deKleer 1993 para una discusión de formas elegantes de construir sistemas de mantenimiento de la verdad y aplicaciones que utilizan el mantenimiento de la verdad. Zabih et al. 1987 describe una extensión no determinista a Scheme que se basa en `amb`; es similar al intérprete descrito en esta sección, pero más sofisticado, porque utiliza el retroceso dirigido por dependencias en lugar del retroceso cronológico. Winston 1992 ofrece una introducción a ambos tipos de retroceso.

48. Nuestro programa usa el siguiente procedimiento para determinar si los elementos de una lista son distintos:

(define (distinct? items) (cond ((null? items) true) ((null? (cdr items)) true) ((member (car items) (cdr items)) false) (else (distinct? (cdr items)))))

`member` es como `memq` excepto que usa `equal?` en lugar de `eq?` para probar la igualdad.

49. Esto está tomado de un folleto llamado "Recreaciones Problemáticas", publicado en la década de 1960 por Litton Industries, donde se atribuye al *Kansas State Engineer*.

50. Aquí usamos la convención de que el primer elemento de cada lista designa la parte de la oración para el resto de las palabras de la lista.

51. Nótese que `parse/word` usa `set!` para modificar la lista de entrada no analizada. Para que esto funcione, nuestro evaluador `amb` debe deshacer los efectos de las operaciones `set!` cuando retrocede.

52. Observe que esta definición es recursiva—un verbo puede ir seguido de

cualquier número de frases preposicionales.

53. Esta técnica es una parte integral del proceso de compilación, que discutiremos en el Capítulo 5. Jonathan Rees escribió un intérprete de Scheme como este alrededor de 1982 para el proyecto T (Rees and Adams 1982). Marc Feeley (1986) (ver también Feeley and Lapalme 1987) inventó independientemente esta técnica en su tesis de maestría.

54. Sin embargo, hay una parte importante de la búsqueda de variables que *puede* hacerse como parte del análisis sintáctico. Como mostraremos en Section 5.5.6, se puede determinar la posición en la estructura del entorno donde se encontrará el valor de la variable, obviando así la necesidad de escanear el entorno para la entrada que coincide con la variable.

55. Ver Ejercicio 4.23 para una idea del procesamiento de secuencias.

56. Snarf: "Agarrar, especialmente un documento o archivo grande con el propósito de usarlo con o sin el permiso del propietario". Snarf down: "Hacer snarf, a veces con la connotación de absorber, procesar o comprender." (Estas definiciones fueron tomadas prestadas de Steele et al. 1983. Ver también Raymond 1993.)

57. La diferencia entre la terminología "perezosa" y la terminología "de orden normal" es algo difusa. Generalmente, "perezoso" se refiere a los mecanismos de evaluadores particulares, mientras que "de orden normal" se refiere a la semántica de los lenguajes, independientemente de cualquier estrategia de evaluación particular. Pero esta no es una distinción rígida, y las dos terminologías a menudo se usan indistintamente.

58. La terminología "estricto" versus "no estricto" significa esencialmente lo mismo que "orden aplicativo" versus "orden normal", excepto que se refiere a procedimientos y argumentos individuales en lugar de al lenguaje en su totalidad. En una conferencia sobre lenguajes de programación, podría escuchar a alguien decir: “El lenguaje de orden normal Hassle tiene ciertas primitivas estrictas. Otros procedimientos toman sus argumentos por evaluación perezosa”

59. La palabra *thunk* fue inventada por un grupo de trabajo informal que estaba discutiendo la implementación de la llamada por nombre en Algol 60. Observaron que la mayor parte del análisis ("pensar en") la expresión podía hacerse en tiempo de compilación; así, en tiempo de ejecución, la expresión ya habría sido "pensada" (Ingerman et al. 1960).

60. Esto es análogo al uso de `force` en los objetos retardados que se introdujeron en el Capítulo 3 para representar flujos. La diferencia crítica

entre lo que estamos haciendo aquí y lo que hicimos en el Capítulo 3 es que estamos incorporando el retraso y el forzamiento en el evaluador, y por lo tanto haciendo esto uniforme y automático en todo el lenguaje.

61. La evaluación perezosa combinada con la memoización a veces se denomina paso de argumentos por *llamada por necesidad*, en contraste con el paso de argumentos por *llamada por nombre*. (La llamada por nombre, introducida en Algol 60, es similar a la evaluación perezosa no memoizada.) Como diseñadores de lenguajes, podemos construir nuestro evaluador para memoizar, no para memoizar, o dejar esto como una opción para los programadores (Ejercicio 4.31). Como podría esperarse del Capítulo 3, estas opciones plantean problemas que se vuelven sutiles y confusos en presencia de asignaciones. (Ver Ejercicio 4.27 y Ejercicio 4.29.) Un excelente artículo de Clinger (1982) intenta aclarar las múltiples dimensiones de confusión que surgen aquí.

62. Nótese que también borramos el `env` del thunk una vez que se ha calculado el valor de la expresión. Esto no supone ninguna diferencia en los valores devueltos por el intérprete. Sin embargo, ayuda a ahorrar espacio, porque eliminar la referencia del thunk al `env` una vez que ya no es necesario permite que esta estructura sea *recolectada como basura* y su espacio reciclado, como discutiremos en Section 5.3.

De manera similar, podríamos haber permitido que los entornos innecesarios en los objetos retardados memoizados de Section 3.5.1 fueran recolectados como basura, haciendo que `memo/proc` hiciera algo como `(set! proc '())` para descartar el procedimiento `proc` (que incluye el entorno en el que se evaluó el `delay`) después de almacenar su valor.

63. Este ejercicio demuestra que la interacción entre la evaluación perezosa y los efectos secundarios puede ser muy confusa. Esto es justo lo que se podría esperar de la discusión en el Capítulo 3.

64. Este es precisamente el problema con el procedimiento `unless`, como en el Ejercicio 4.26.

65. Esta es la representación procedimental descrita en el Ejercicio 2.4. Esencialmente, cualquier representación procedimental (por ejemplo, una implementación de paso de mensajes) funcionaría igual de bien. Nótese que podemos instalar estas definiciones en el evaluador perezoso simplemente escribiéndolas en el bucle del controlador. Si originalmente hubiéramos incluido `cons`, `car` y `cdr` como primitivas en el entorno global, se redefinirán. (También ver Ejercicio 4.33 y Ejercicio 4.34.)

66. Esto nos permite crear versiones retardadas de tipos más generales de

estructuras de lista, no solo secuencias. Hughes 1990 discute algunas aplicaciones de los "árboles perezosos".

67. Asumimos que hemos definido previamente un procedimiento `prime?` que prueba si los números son primos. Incluso con `prime?` definido, el procedimiento `prime/sum/pair` puede parecer sospechosamente similar al inútil intento de "pseudo-Lisp" de definir la función de raíz cuadrada, que describimos al principio de Section 1.1.7. De hecho, un procedimiento de raíz cuadrada en esa línea puede formularse realmente como un programa no determinista. Al incorporar un mecanismo de búsqueda en el evaluador, estamos erosionando la distinción entre descripciones puramente declarativas y especificaciones imperativas de cómo calcular las respuestas. Iremos aún más lejos en esta dirección en Section 4.4.

68. La idea de `amb` para la programación no determinista fue descrita por primera vez en 1961 por John McCarthy (ver McCarthy 1963).

69. En realidad, la distinción entre devolver no determinísticamente una sola elección y devolver todas las elecciones depende en cierto modo de nuestro punto de vista. Desde la perspectiva del código que utiliza el valor, la elección no determinista devuelve un solo valor. Desde la perspectiva del programador que diseña el código, la elección no determinista devuelve potencialmente todos los valores posibles, y el cálculo se ramifica de modo que cada valor se investiga por separado.

70. Se podría objetar que este es un mecanismo irremediablemente ineficiente. Podría requerir millones de procesadores para resolver algún problema fácilmente planteado de esta manera, y la mayor parte del tiempo la mayoría de esos procesadores estarían inactivos. Esta objeción debe tomarse en el contexto de la historia. La memoria solía considerarse un producto caro. En 1964 un megabyte de RAM costaba alrededor de $400,000. Ahora cada computadora personal tiene muchos megabytes de RAM, y la mayor parte del tiempo la mayor parte de esa RAM no se utiliza. Es difícil subestimar el costo de la electrónica producida en masa.

71. Automágicamente: "Automáticamente, pero de una manera que, por alguna razón (típicamente porque es demasiado complicado, o demasiado feo, o quizás incluso demasiado trivial), el hablante no tiene ganas de explicar". (Steele et al. 1983, Raymond 1993)

72. La integración de estrategias de búsqueda automática en los lenguajes de programación ha tenido una historia larga y accidentada. Las primeras sugerencias de que los algoritmos no deterministas podrían codificarse elegantemente en un lenguaje de programación con búsqueda y retroceso

automático provinieron de Robert Floyd (1967). Carl Hewitt (1969) inventó un lenguaje de programación llamado Planner que soportaba explícitamente el retroceso cronológico automático, proporcionando una estrategia de búsqueda en profundidad incorporada. Sussman et al. (1971) implementaron un subconjunto de este lenguaje, llamado MicroPlanner, que se utilizó para apoyar el trabajo en la resolución de problemas y la planificación de robots. Ideas similares, surgidas de la lógica y la demostración de teoremas, llevaron a la génesis en Edimburgo y Marsella del elegante lenguaje Prolog (que discutiremos en Section 4.4). Después de una frustración suficiente con la búsqueda automática, McDermott y Sussman (1972) desarrollaron un lenguaje llamado Conniver, que incluía mecanismos para poner la estrategia de búsqueda bajo el control del programador. Sin embargo, esto resultó engorroso, y Sussman y Stallman 1975 encontraron un enfoque más manejable mientras investigaban métodos de análisis simbólico para circuitos eléctricos. Desarrollaron un esquema de retroceso no cronológico que se basaba en rastrear las dependencias lógicas que conectan los hechos, una técnica que ha llegado a conocerse como *retroceso dirigido por dependencias*. Aunque su método era complejo, produjo programas razonablemente eficientes porque realizaba poca búsqueda redundante. Doyle (1979) y McAllester (1978; 1980) generalizaron y clarificaron los métodos de Stallman y Sussman, desarrollando un nuevo paradigma para formular la búsqueda que ahora se llama *mantenimiento de la verdad*. Todos los sistemas modernos de resolución de problemas utilizan alguna forma de sistema de mantenimiento de la verdad como sustrato. Ver Forbus y deKleer 1993 para una discusión de formas elegantes de construir sistemas de mantenimiento de la verdad y aplicaciones que utilizan el mantenimiento de la verdad. Zabih et al. 1987 describe una extensión no determinista a Scheme que se basa en `amb`; es similar al intérprete descrito en esta sección, pero más sofisticado, porque utiliza el retroceso dirigido por dependencias en lugar del retroceso cronológico. Winston 1992 ofrece una introducción a ambos tipos de retroceso.

73. Nuestro programa usa el siguiente procedimiento para determinar si los elementos de una lista son distintos:

(define (distinct? items) (cond ((null? items) true) ((null? (cdr items)) true) ((member (car items) (cdr items)) false) (else (distinct? (cdr items)))))

`member` es como `memq` excepto que usa `equal?` en lugar de `eq?` para probar la igualdad.

74. Esto está tomado de un folleto llamado "Recreaciones Problemáticas", publicado en la década de 1960 por Litton Industries, donde se atribuye al *Kansas State Engineer*.

75. Aquí usamos la convención de que el primer elemento de cada lista designa la parte de la oración para el resto de las palabras de la lista.

76. Nótese que `parse/word` usa `set!` para modificar la lista de entrada no analizada. Para que esto funcione, nuestro evaluador `amb` debe deshacer los efectos de las operaciones `set!` cuando retrocede.

77. Observe que esta definición es recursiva—un verbo puede ir seguido de cualquier número de frases preposicionales.

78. Este tipo de gramática puede volverse arbitrariamente complejo, pero es solo un juguete en lo que respecta a la comprensión real del lenguaje. La comprensión real del lenguaje natural por computadora requiere una mezcla elaborada de análisis sintáctico e interpretación del significado. Por otro lado, incluso los analizadores de juguete pueden ser útiles para soportar lenguajes de comandos flexibles para programas como los sistemas de recuperación de información. Winston 1992 discute enfoques computacionales para la comprensión real del lenguaje y también las aplicaciones de gramáticas simples a los lenguajes de comandos.

79. Aunque la idea de Alyssa funciona perfectamente (y es sorprendentemente simple), las oraciones que genera son un poco aburridas —no muestrean las posibles oraciones de este lenguaje de una manera muy interesante. De hecho, la gramática es altamente recursiva en muchos lugares, y la técnica de Alyssa "cae en" una de estas recursiones y se atasca. Vea el Ejercicio 4.50 para una forma de lidiar con esto.

80. Elegimos implementar el evaluador perezoso en Section 4.2 como una modificación del evaluador metacircular ordinario de Section 4.1.1. En contraste, basaremos el evaluador `amb` en el evaluador de análisis de Section 4.1.7, porque los procedimientos de ejecución en ese evaluador proporcionan un marco conveniente para implementar el retroceso.

81. Asumimos que el evaluador soporta `let` (ver Ejercicio 4.22), que hemos usado en nuestros programas no deterministas.

82. No nos preocupamos por deshacer las definiciones, ya que podemos asumir que las definiciones internas se escanean (Section 4.1.6).

83. La programación lógica ha surgido de una larga historia de investigación en la demostración automática de teoremas. Los primeros programas de demostración de teoremas podían lograr muy poco, porque buscaban exhaustivamente en el espacio de las posibles pruebas. El gran avance que hizo plausible tal búsqueda fue el descubrimiento a principios de la década de 1960 del *algoritmo de unificación* y el *principio de resolución* (Robinson

1965). La resolución fue utilizada, por ejemplo, por Green y Raphael (1968) (ver también Green 1969) como la base para un sistema deductivo de respuesta a preguntas. Durante la mayor parte de este período, los investigadores se concentraron en algoritmos que garantizan encontrar una prueba si existe. Tales algoritmos eran difíciles de controlar y dirigir hacia una prueba. Hewitt (1969) reconoció la posibilidad de fusionar la estructura de control de un lenguaje de programación con las operaciones de un sistema de manipulación lógica, lo que llevó al trabajo en la búsqueda automática mencionado en Section 4.3.1 (Nota 4.47). Al mismo tiempo que se hacía esto, Colmerauer, en Marsella, estaba desarrollando sistemas basados en reglas para manipular el lenguaje natural (ver Colmerauer et al. 1973). Inventó un lenguaje de programación llamado Prolog para representar esas reglas. Kowalski (1973; 1979), en Edimburgo, reconoció que la ejecución de un programa Prolog podía interpretarse como la demostración de teoremas (usando una técnica de prueba llamada resolución de cláusulas de Horn lineales). La fusión de las dos últimas corrientes condujo al movimiento de la programación lógica. Así, al asignar el crédito por el desarrollo de la programación lógica, los franceses pueden señalar la génesis de Prolog en la Universidad de Marsella, mientras que los británicos pueden destacar el trabajo en la Universidad de Edimburgo. Según la gente de MIT, la programación lógica fue desarrollada por estos grupos en un intento de averiguar de qué estaba hablando Hewitt en su brillante pero impenetrable tesis de doctorado. Para una historia de la programación lógica, ver Robinson 1983.

84. Para ver la correspondencia entre las reglas y el procedimiento, sea `x` en el procedimiento (donde `x` no es vacío) correspondiente a `(cons u v)` en la regla. Entonces `z` en la regla corresponde a la `concatenación` de `(cdr x)` y `y`.

85. Esto ciertamente no libera al usuario de todo el problema de cómo calcular la respuesta. Hay muchos conjuntos de reglas matemáticamente equivalentes diferentes para formular la relación `append`, solo algunos de los cuales pueden convertirse en dispositivos efectivos para calcular en cualquier dirección. Además, a veces la información de "qué es" no da ninguna pista de "cómo" calcular una respuesta. Por ejemplo, considere el problema de calcular la y tal que $y^2 = x$.

86. El interés en la programación lógica alcanzó su punto máximo a principios de los años 80 cuando el gobierno japonés inició un ambicioso proyecto destinado a construir supercomputadoras optimizadas para ejecutar lenguajes de programación lógica. La velocidad de tales computadoras se mediría en LIPS (Inferencias Lógicas por Segundo) en lugar de los habituales FLOPS (Operaciones de Punto Flotante por Segundo). Aunque el proyecto

tuvo éxito en el desarrollo de hardware y software como se planeó originalmente, la industria informática internacional se movió en una dirección diferente. Ver Feigenbaum y Shrobe 1993 para una evaluación general del proyecto japonés. La comunidad de programación lógica también ha avanzado para considerar la programación relacional basada en técnicas distintas de la simple coincidencia de patrones, como la capacidad de lidiar con restricciones numéricas como las ilustradas en el sistema de propagación de restricciones de Section 3.3.5.

87. Esto usa la notación de cola punteada introducida en el Ejercicio 2.20.

88. En realidad, esta descripción de `not` es válida solo para casos simples. El comportamiento real de `not` es más complejo. Examinaremos las peculiaridades de `not` en las secciones Sección 4.4.2 y Sección 4.4.3.

89. `lisp/value` debería usarse solo para realizar una operación no proporcionada en el lenguaje de consulta. En particular, no debería usarse para probar la igualdad (ya que para eso está diseñada la coincidencia en el lenguaje de consulta) o la desigualdad (ya que eso se puede hacer con la regla `same` que se muestra a continuación).

90. Nótese que no necesitamos `same` para hacer que dos cosas sean iguales: simplemente usamos la misma variable de patrón para cada una—en efecto, tenemos una cosa en lugar de dos en primer lugar. Por ejemplo, vea `?town` en la regla `lives/near` y `?middle/manager` en la regla `wheel` a continuación. `same` es útil cuando queremos forzar a dos cosas a ser diferentes, como `?person/1` y `?person/2` en la regla `lives/near`. Aunque usar la misma variable de patrón en dos partes de una consulta fuerza a que aparezca el mismo valor en ambos lugares, usar diferentes variables de patrón no fuerza a que aparezcan valores diferentes. (Los valores asignados a diferentes variables de patrón pueden ser iguales o diferentes).

91. También permitiremos reglas sin cuerpo, como en `same`, y interpretaremos tal regla como que la conclusión de la regla se satisface con cualesquiera valores de las variables.

92. Debido a que la coincidencia es generalmente muy costosa, nos gustaría evitar aplicar el coincidente completo a cada elemento de la base de datos. Esto generalmente se arregla dividiendo el proceso en una coincidencia rápida y gruesa y la coincidencia final. La coincidencia gruesa filtra la base de datos para producir un pequeño conjunto de candidatos para la coincidencia final. Con cuidado, podemos organizar nuestra base de datos para que parte del trabajo de la coincidencia gruesa pueda hacerse cuando se construye la base de datos en lugar de cuando queremos seleccionar los

candidatos. Esto se llama *indexar* la base de datos. Existe una vasta tecnología construida en torno a los esquemas de indexación de bases de datos. Nuestra implementación, descrita en Section 4.4.4, contiene una forma simplista de tal optimización.

93. Pero este tipo de explosión exponencial no es común en las consultas `y` porque las condiciones agregadas tienden a reducir en lugar de expandir el número de marcos producidos.

94. Hay una gran literatura sobre sistemas de gestión de bases de datos que se ocupa de cómo manejar consultas complejas de manera eficiente.

95. Hay una diferencia sutil entre esta implementación de filtro de `not` y el significado usual de `not` en la lógica matemática. Ver Section 4.4.3.

96. En la coincidencia de patrones unilateral, todas las ecuaciones que contienen variables de patrón son explícitas y ya están resueltas para el desconocido (la variable de patrón).

97. Otra forma de pensar en la unificación es que genera el patrón más general que es una especialización de los dos patrones de entrada. Es decir, la unificación de `(?x a)` y `((b ?y) ?z)` es `((b ?y) a)`, y la unificación de `(?x a ?y)` y `(?y ?z a)`, discutida anteriormente, es `(a a a)`. Para nuestra implementación, es más conveniente pensar en el resultado de la unificación como un marco en lugar de un patrón.

98. Dado que la unificación es una generalización de la coincidencia, podríamos simplificar el sistema usando el unificador para producir ambos flujos. Sin embargo, tratar el caso fácil con el comparador simple ilustra cómo la coincidencia (a diferencia de la unificación completa) puede ser útil por sí misma.

99. La razón por la que usamos flujos (en lugar de listas) de marcos es que la aplicación recursiva de reglas puede generar un número infinito de valores que satisfacen una consulta. La evaluación retardada encarnada en los flujos es crucial aquí: el sistema imprimirá las respuestas una por una a medida que se generan, independientemente de si hay un número finito o infinito de respuestas.

100. Que un método particular de inferencia sea legítimo no es una afirmación trivial. Se debe probar que si se comienza con premisas verdaderas, solo se pueden derivar conclusiones verdaderas. El método de inferencia representado por las aplicaciones de reglas es el *modus ponens*, el método familiar de inferencia que dice que si *A* es verdadero y *A implica B* es verdadero, entonces podemos concluir que *B* es verdadero.

101. Debemos calificar esta afirmación acordando que, al hablar de la "inferencia" lograda por un programa lógico, asumimos que el cálculo termina. Desafortunadamente, incluso esta afirmación calificada es falsa para nuestra implementación del lenguaje de consulta (y también falsa para los programas en Prolog y la mayoría de los otros lenguajes de programación lógica actuales) debido a nuestro uso de `not` y `lisp/value`. Como describiremos a continuación, el `not` implementado en el lenguaje de consulta no siempre es consistente con el `not` de la lógica matemática, y `lisp/value` introduce complicaciones adicionales. Podríamos implementar un lenguaje consistente con la lógica matemática simplemente eliminando `not` y `lisp/value` del lenguaje y acordando escribir programas usando solo consultas simples, `y` y `o`. Sin embargo, esto restringiría en gran medida el poder expresivo del lenguaje. Una de las principales preocupaciones de la investigación en programación lógica es encontrar formas de lograr una mayor consistencia con la lógica matemática sin sacrificar indebidamente el poder expresivo.

102. Esto no es un problema de la lógica, sino de la interpretación procedimental de la lógica proporcionada por nuestro intérprete. Podríamos escribir un intérprete que no cayera en un bucle aquí. Por ejemplo, podríamos enumerar todas las pruebas derivables de nuestras aserciones y nuestras reglas en un orden de búsqueda en anchura en lugar de en profundidad. Sin embargo, tal sistema hace que sea más difícil aprovechar el orden de las deducciones en nuestros programas. Un intento de incorporar un control sofisticado en tal programa se describe en deKleer et al. 1977. Otra técnica, que no conduce a problemas de control tan serios, es introducir conocimiento especial, como detectores para tipos particulares de bucles (Ejercicio 4.67). Sin embargo, no puede haber un esquema general para evitar de manera confiable que un sistema caiga en caminos infinitos al realizar deducciones. Imagine una regla diabólica de la forma "Para mostrar que $P(x)$ es verdadero, demuestre que $P(f(x))$ es verdadero", para alguna función f elegida adecuadamente.

103. Considere la consulta `(not (baseball/fan (Bitdiddle Ben)))`. El sistema encuentra que `(baseball/fan (Bitdiddle Ben))` no está en la base de datos, por lo que el marco vacío no satisface el patrón y no se filtra del flujo inicial de marcos. El resultado de la consulta es por lo tanto el marco vacío, que se usa para instanciar la consulta de entrada para producir `(not (baseball/fan (Bitdiddle Ben)))`.

104. Una discusión y justificación de este tratamiento de `not` puede encontrarse en el artículo de Clark (1978).

105. En general, unificar `?y` con una expresión que involucra a `?y` requeriría

que pudiéramos encontrar un punto fijo de la ecuación `?y` = ⟨*expresión que involucra a* `?y`⟩. A veces es posible formar sintácticamente una expresión que parece ser la solución. Por ejemplo, `?y` = `(f ?y)` parece tener el punto fijo `(f (f (f ... )))`, que podemos producir comenzando con la expresión `(f ?y)` y sustituyendo repetidamente `(f ?y)` por `?y`. Desafortunadamente, no toda ecuación de este tipo tiene un punto fijo significativo. Los problemas que surgen aquí son similares a los problemas de manipulación de series infinitas en matemáticas. Por ejemplo, sabemos que 2 es la solución a la ecuación $y = 1 + y/2$. Comenzando con la expresión $1 + y/2$ y sustituyendo repetidamente $1 + y/2$ por y se obtiene

$$2 = y = 1 + {y \over 2} = 1 + {1\over2}\left(1 + {y \over 2}\right)
= 1 + {1\over2} + {y \over 4} = \dots ,$$

lo que lleva a

$$2 = 1 + {1\over2} + {1\over4} + {1\over8} + \dots.$$

Sin embargo, si intentamos la misma manipulación comenzando con la observación de que -1 es la solución a la ecuación $y = 1 + 2y$, obtenemos

$$-1 = y = 1 + 2y = 1 + 2(1 + 2y) = 1 + 2 + 4y = \ldots,$$

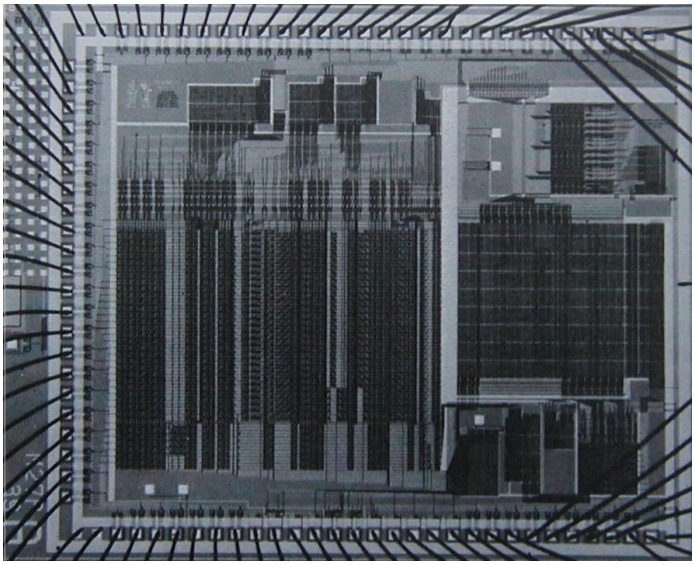
lo que lleva a

$$-1 = 1 + 2 + 4 + 8 + \ldots.$$

Aunque las manipulaciones formales utilizadas para derivar estas dos ecuaciones son idénticas, el primer resultado es una afirmación válida sobre las series infinitas, pero el segundo no lo es. De manera similar, para nuestros resultados de unificación, razonar con una expresión construida sintácticamente arbitrariamente puede llevar a errores.

106. La mayoría de los sistemas Lisp le dan al usuario la capacidad de modificar el procedimiento `read` ordinario para realizar tales transformaciones definiendo *caracteres de macro de lector*. Las expresiones entrecomilladas ya se manejan de esta manera: el lector traduce automáticamente `'expresión` a `(quote expresión)` antes de que el evaluador la vea. Podríamos hacer arreglos para que `?expresión` se transforme en `(? expresión)` de la misma manera; sin embargo, en aras de la claridad, hemos incluido el procedimiento de transformación aquí explícitamente.

`expand/question/mark` y `contract/question/mark` usan varios procedimientos con `string` en sus nombres. Estas son primitivas de Scheme.



Computación con Máquinas de Registros

> Mi objetivo es mostrar que la máquina celestial no es una especie de ser divino y vivo, sino una especie de mecanismo de relojería (y quien cree que un reloj tiene alma atribuye la gloria del creador a la obra), en la medida en que casi todos los múltiples movimientos son causados por una fuerza muy simple y material, así como todos los movimientos del reloj son causados por un solo peso.
>
> —Johannes Kepler (carta a Herwart von Hohenburg, 1605)

COMENZAMOS ESTE LIBRO estudiando los procesos y describiéndolos en términos de procedimientos escritos en Lisp. Para explicar el significado de estos procedimientos, utilizamos una sucesión de modelos de evaluación: el modelo de sustitución del Capítulo 1, el modelo de entorno del Capítulo 3 y el evaluador metacircular del Capítulo 4. Nuestro examen del evaluador metacircular, en particular, disipó gran parte del misterio de cómo se interpretan los lenguajes tipo Lisp. Pero incluso el evaluador metacircular deja importantes preguntas sin respuesta, porque no logra elucidar los mecanismos de control en un sistema Lisp. Por ejemplo, el evaluador no explica cómo la evaluación de una subexpresión logra devolver un valor a la expresión que utiliza este valor, ni el evaluador explica cómo algunos procedimientos recursivos generan procesos iterativos (es decir, se evalúan utilizando un espacio constante) mientras que otros procedimientos recursivos generan procesos recursivos. Estas preguntas quedan sin respuesta porque el evaluador metacircular es en sí mismo un programa Lisp y, por lo tanto, hereda la estructura de control del sistema Lisp subyacente. Para proporcionar una descripción más completa de la estructura de control del evaluador de Lisp, debemos trabajar a un nivel más primitivo que el propio Lisp.

En este capítulo describiremos los procesos en términos de la operación paso a paso de una computadora tradicional. Tal computadora, o *máquina de registros*, ejecuta secuencialmente *instrucciones* que manipulan el contenido de un conjunto fijo de elementos de almacenamiento llamados *registros*. Una

instrucción típica de una máquina de registros aplica una operación primitiva al contenido de algunos registros y asigna el resultado a otro registro. Nuestras descripciones de los procesos ejecutados por las máquinas de registros se parecerán mucho a los programas en "lenguaje de máquina" de las computadoras tradicionales. Sin embargo, en lugar de centrarnos en el lenguaje de máquina de cualquier computadora en particular, examinaremos varios procedimientos de Lisp y diseñaremos una máquina de registros específica para ejecutar cada procedimiento. Así, abordaremos nuestra tarea desde la perspectiva de un arquitecto de hardware en lugar de la de un programador de computadoras en lenguaje de máquina. En el diseño de máquinas de registros, desarrollaremos mecanismos para implementar importantes construcciones de programación como la recursión. También presentaremos un lenguaje para describir diseños de máquinas de registros. En Section 5.2 implementaremos un programa de Lisp que utiliza estas descripciones para simular las máquinas que diseñamos.

La mayoría de las operaciones primitivas de nuestras máquinas de registros son muy simples. Por ejemplo, una operación podría sumar los números obtenidos de dos registros, produciendo un resultado que se almacenará en un tercer registro. Tal operación puede realizarse mediante hardware fácilmente descrito. Sin embargo, para tratar con la estructura de listas, también utilizaremos las operaciones de memoria `car`, `cdr` y `cons`, que requieren un elaborado mecanismo de asignación de almacenamiento. En Section 5.3 estudiaremos su implementación en términos de operaciones más elementales.

En Section 5.4, después de haber acumulado experiencia en la formulación de procedimientos simples como máquinas de registros, diseñaremos una máquina que lleve a cabo el algoritmo descrito por el evaluador metacircular de Section 4.1. Esto llenará el vacío en nuestra comprensión de cómo se interpretan las expresiones de Scheme, proporcionando un modelo explícito para los mecanismos de control en el evaluador. En Section 5.5 estudiaremos un compilador simple que traduce programas de Scheme a secuencias de instrucciones que pueden ejecutarse directamente con los registros y operaciones de la máquina de registros del evaluador.

Diseñando Máquinas de Registros

Para diseñar una máquina de registros, debemos diseñar sus *rutas de datos* (registros y operaciones) y el *controlador* que secuencia estas operaciones. Para ilustrar el diseño de una máquina de registros simple, examinemos el Algoritmo de Euclides, que se utiliza para calcular el máximo común divisor

(MCD) de dos enteros. Como vimos en Section 1.2.5, el Algoritmo de Euclides puede llevarse a cabo mediante un proceso iterativo, como lo especifica el siguiente procedimiento:

```
(define (gcd a b) (if (= b 0) a (gcd b (remainder a b))))
```

Una máquina para llevar a cabo este algoritmo debe llevar la cuenta de dos números, *a* y *b*, así que asumamos que estos números están almacenados en dos registros con esos nombres. Las operaciones básicas requeridas son probar si el contenido del registro `b` es cero y calcular el resto del contenido del registro `a` dividido por el contenido del registro `b`. La operación de resto es un proceso complejo, pero supongamos por el momento que tenemos un dispositivo primitivo que calcula restos. En cada ciclo del algoritmo del MCD, el contenido del registro `a` debe ser reemplazado por el contenido del registro `b`, y el contenido de `b` debe ser reemplazado por el resto del contenido anterior de `a` dividido por el contenido anterior de `b`. Sería conveniente si estos reemplazos pudieran hacerse simultáneamente, pero en nuestro modelo de máquinas de registros asumiremos que solo un registro puede recibir un nuevo valor en cada paso. Para lograr los reemplazos, nuestra máquina utilizará un tercer registro "temporal", que llamaremos `t`. (Primero se colocará el resto en `t`, luego el contenido de `b` se colocará en `a`, y finalmente el resto almacenado en `t` se colocará en `b`.)

Podemos ilustrar los registros y las operaciones requeridas para esta máquina utilizando el diagrama de ruta de datos que se muestra en la Figura 5.1. En este diagrama, los registros (`a`, `b` y `t`) se representan mediante rectángulos. Cada forma de asignar un valor a un registro se indica con una flecha con una `X` detrás de la cabeza, que apunta desde la fuente de datos hasta el registro. Podemos pensar en la `X` como un botón que, al ser presionado, permite que el valor en la fuente "fluya" hacia el registro designado. La etiqueta junto a cada botón es el nombre que usaremos para referirnos al botón. Los nombres son arbitrarios y pueden elegirse para que tengan valor mnemotécnico (por ejemplo, `a←b` denota presionar el botón que asigna el contenido del registro `b` al registro `a`). La fuente de datos para un registro puede ser otro registro (como en la asignación `a←b`), el resultado de una operación (como en la asignación `t←r`), o una constante (un valor incorporado que no se puede cambiar, representado en un diagrama de ruta de datos por un triángulo que contiene la constante).

Figura 5.1: Rutas de datos para una máquina GCD.

Una operación que calcula un valor a partir de constantes y el contenido de

los registros se representa en un diagrama de ruta de datos mediante un trapecio que contiene un nombre para la operación. Por ejemplo, el cuadro marcado como `rem` en la Figura 5.1 representa una operación que calcula el resto del contenido de los registros `a` y `b` a los que está conectado. Flechas (sin botones) apuntan desde los registros de entrada y las constantes al cuadro, y las flechas conectan el valor de salida de la operación a los registros. Una prueba se representa mediante un círculo que contiene un nombre para la prueba. Por ejemplo, nuestra máquina GCD tiene una operación que prueba si el contenido del registro `b` es cero. Una prueba también tiene flechas desde sus registros de entrada y constantes, pero no tiene flechas de salida; su valor es utilizado por el controlador en lugar de por las rutas de datos. En general, el diagrama de ruta de datos muestra los registros y las operaciones que se requieren para la máquina y cómo deben conectarse. Si vemos las flechas como cables y los botones `x` como interruptores, el diagrama de ruta de datos es muy parecido al diagrama de cableado de una máquina que podría construirse a partir de componentes eléctricos.

Figura 5.2: Controlador para una máquina GCD.

Para que las rutas de datos realmente calculen los MCDs, los botones deben presionarse en la secuencia correcta. Describiremos esta secuencia en términos de un diagrama de controlador, como se ilustra en la Figura 5.2. Los elementos del diagrama de controlador indican cómo deben operarse los componentes de la ruta de datos. Los cuadros rectangulares en el diagrama de controlador identifican los botones de la ruta de datos que se deben presionar, y las flechas describen la secuencia de un paso al siguiente. El rombo en el diagrama representa una decisión. Se seguirá una de las dos flechas de secuencia, dependiendo del valor de la prueba de la ruta de datos identificada en el rombo. Podemos interpretar el controlador en términos de una analogía física: piense en el diagrama como un laberinto en el que rueda una canica. Cuando la canica entra en un cuadro, presiona el botón de la ruta de datos que nombra el cuadro. Cuando la canica entra en un nodo de decisión (como la prueba de `b` = 0), sale del nodo por la ruta determinada por el resultado de la prueba indicada. En conjunto, las rutas de datos y el controlador describen completamente una máquina para calcular MCDs. Iniciamos el controlador (la canica rodante) en el lugar marcado como `inicio`, después de colocar números en los registros `a` y `b`. Cuando el controlador llegue a `hecho`, encontraremos el valor del MCD en el registro `a`.

Ejercicio 5.1: Diseñe una máquina de registros para calcular factoriales usando el algoritmo iterativo especificado por el siguiente procedimiento.

Dibuje diagramas de ruta de datos y de controlador para esta máquina.

(define (factorial n) (define (iter product counter) (if (> counter n) product (iter (* counter product) (+ counter 1)))) (iter 1 1))

Un Lenguaje para Describir Máquinas de Registros

Los diagramas de ruta de datos y de controlador son adecuados para representar máquinas simples como el MCD, pero son engorrosos para describir máquinas grandes como un intérprete de Lisp. Para que sea posible tratar con máquinas complejas, crearemos un lenguaje que presente, en forma textual, toda la información dada por los diagramas de ruta de datos y de controlador. Comenzaremos con una notación que refleje directamente los diagramas.

Definimos las rutas de datos de una máquina describiendo los registros y las operaciones. Para describir un registro, le damos un nombre y especificamos los botones que controlan la asignación a él. Le damos a cada uno de estos botones un nombre y especificamos la fuente de los datos que ingresan al registro bajo el control del botón. (La fuente es un registro, una constante o una operación). Para describir una operación, le damos un nombre y especificamos sus entradas (registros o constantes).

Definimos el controlador de una máquina como una secuencia de *instrucciones* junto con *etiquetas* que identifican *puntos de entrada* en la secuencia. Una instrucción es una de las siguientes:

- El nombre de un botón de la ruta de datos que se debe presionar para asignar un valor a un registro. (Esto corresponde a un cuadro en el diagrama del controlador).
- Una instrucción `test`, que realiza una prueba especificada.
- Una bifurcación condicional (instrucción `branch`) a una ubicación indicada por una etiqueta del controlador, basada en el resultado de la prueba anterior. (La prueba y la bifurcación juntas corresponden a un rombo en el diagrama del controlador). Si la prueba es falsa, el controlador debe continuar con la siguiente instrucción de la secuencia. De lo contrario, el controlador debe continuar con la instrucción después de la etiqueta.
- Una bifurcación incondicional (instrucción `goto`) que nombra una etiqueta del controlador en la que continuar la ejecución.

La máquina comienza al principio de la secuencia de instrucciones del controlador y se detiene cuando la ejecución llega al final de la secuencia. Excepto cuando una bifurcación cambia el flujo de control, las instrucciones se ejecutan en el orden en que están listadas.

Figura 5.3: ↓ Una especificación de la máquina GCD.

```
(data-paths (registers ((name a) (buttons ((name a<-b) (source (register
b))))) ((name b) (buttons ((name b<-t) (source (register t))))) ((name t)
(buttons ((name t<-r) (source (operation rem)))))) (operations ((name
rem) (inputs (register a) (register b))) ((name =) (inputs (register b)
(constant 0))))) (controller test-b  ; etiqueta  (test =)  ; prueba  (branch
(label gcd-done))  ; bifurcación condicional  (t<-r)  ; pulsación de botón
(a<-b)  ; pulsación de botón  (b<-t)  ; pulsación de botón  (goto (label
test-b))  ; bifurcación incondicional  gcd-done)  ; etiqueta
```

La Figura 5.3 muestra la máquina GCD descrita de esta manera. Este ejemplo solo insinúa la generalidad de estas descripciones, ya que la máquina GCD es un caso muy simple: cada registro tiene solo un botón, y cada botón y prueba se usa solo una vez en el controlador.

Desafortunadamente, es difícil leer tal descripción. Para entender las instrucciones del controlador debemos referirnos constantemente a las definiciones de los nombres de los botones y los nombres de las operaciones, y para entender lo que hacen los botones podemos tener que referirnos a las definiciones de los nombres de las operaciones. Por lo tanto, transformaremos nuestra notación para combinar la información de las descripciones de la ruta de datos y el controlador para que la veamos toda junta.

Para obtener esta forma de descripción, reemplazaremos los nombres arbitrarios de botones y operaciones por las definiciones de su comportamiento. Es decir, en lugar de decir (en el controlador) "Presionar el botón `t←r`" y decir por separado (en las rutas de datos) "El botón `t←r` asigna el valor de la operación `rem` al registro `t`" y "Las entradas de la operación `rem` son los contenidos de los registros `a` y `b`", diremos (en el controlador) "Presionar el botón que asigna al registro `t` el valor de la operación `rem` sobre los contenidos de los registros `a` y `b`". De manera similar, en lugar de decir (en el controlador) "Realizar la prueba `=`" y decir por separado (en las rutas de datos) "La prueba `=` opera sobre el contenido del registro `b` y la constante 0", diremos "Realizar la prueba `=` sobre el contenido del registro `b` y la constante 0". Omitiremos la descripción de la ruta de datos, dejando solo la secuencia del controlador. Así, la máquina GCD se describe de la siguiente manera:

(controller test-b (test (op =) (reg b) (const 0)) (branch (label gcd-done)) (assign t (op rem) (reg a) (reg b)) (assign a (reg b)) (assign b (reg t)) (goto (label test-b)) gcd-done)

Esta forma de descripción es más fácil de leer que la ilustrada en la Figura 5.3, pero también tiene desventajas:

- Es más verboso para máquinas grandes, porque las descripciones completas de los elementos de la ruta de datos se repiten cada vez que se mencionan los elementos en la secuencia de instrucciones del controlador. (Esto no es un problema en el ejemplo de GCD, porque cada operación y botón se usa solo una vez.) Además, repetir las descripciones de la ruta de datos oscurece la estructura real de la ruta de datos de la máquina; no es obvio para una máquina grande cuántos registros, operaciones y botones hay y cómo están interconectados.

- Debido a que las instrucciones del controlador en la definición de una máquina se parecen a las expresiones de Lisp, es fácil olvidar que no son expresiones de Lisp arbitrarias. Solo pueden denotar operaciones de máquina legales. Por ejemplo, las operaciones solo pueden operar directamente sobre constantes y el contenido de los registros, no sobre los resultados de otras operaciones.

A pesar de estas desventajas, usaremos este lenguaje de máquina de registros a lo largo de este capítulo, porque nos preocuparemos más por entender los controladores que por entender los elementos y las conexiones en las rutas de datos. Sin embargo, debemos tener en cuenta que el diseño de la ruta de datos es crucial en el diseño de máquinas reales.

Ejercicio 5.2: Use el lenguaje de máquina de registros para describir la máquina factorial iterativa del Ejercicio 5.1.

Acciones

Modifiquemos la máquina GCD para que podamos escribir los números cuyo MCD queremos y obtener la respuesta impresa en nuestra terminal. No discutiremos cómo hacer una máquina que pueda leer e imprimir, pero asumiremos (como lo hacemos cuando usamos `read` y `display` en Scheme) que están disponibles como operaciones primitivas.[1]

Figura 5.4: Una máquina GCD que lee entradas e imprime resultados.

`read` es como las operaciones que hemos estado usando en el sentido de que

produce un valor que puede almacenarse en un registro. Pero `read` no toma entradas de ningún registro; su valor depende de algo que sucede fuera de las partes de la máquina que estamos diseñando. Permitiremos que las operaciones de nuestra máquina tengan tal comportamiento, y por lo tanto dibujaremos y anotaremos el uso de `read` al igual que hacemos con cualquier otra operación que calcule un valor.

`print`, por otro lado, difiere de las operaciones que hemos estado usando de una manera fundamental: no produce un valor de salida para ser almacenado en un registro. Aunque tiene un efecto, este efecto no está en una parte de la máquina que estamos diseñando. Nos referiremos a este tipo de operación como una *acción*. Representaremos una acción en un diagrama de ruta de datos tal como representamos una operación que calcula un valor—como un trapecio que contiene el nombre de la acción. Las flechas apuntan al cuadro de acción desde cualquier entrada (registros o constantes). También asociamos un botón con la acción. Presionar el botón hace que la acción suceda. Para hacer que un controlador presione un botón de acción usamos un nuevo tipo de instrucción llamada `perform`. Así, la acción de imprimir el contenido del registro `a` se representa en una secuencia del controlador por la instrucción

```
(perform (op print) (reg a))
```

La Figura 5.4 muestra las rutas de datos y el controlador para la nueva máquina MCD. En lugar de que la máquina se detenga después de imprimir la respuesta, la hemos hecho comenzar de nuevo, para que lea repetidamente un par de números, calcule su MCD e imprima el resultado. Esta estructura es como los bucles del controlador que usamos en los intérpretes del Capítulo 4.

Abstracción en el Diseño de Máquinas

A menudo definiremos una máquina para incluir operaciones "primitivas" que en realidad son muy complejas. Por ejemplo, en Section 5.4 y Section 5.5 trataremos las manipulaciones del entorno de Scheme como primitivas. Tal abstracción es valiosa porque nos permite ignorar los detalles de las partes de una máquina para que podamos concentrarnos en otros aspectos del diseño. El hecho de que hayamos barrido mucha complejidad bajo la alfombra, sin embargo, no significa que un diseño de máquina sea irreal. Siempre podemos reemplazar las "primitivas" complejas por operaciones primitivas más simples.

Considere la máquina GCD. La máquina tiene una instrucción que calcula el resto del contenido de los registros `a` y `b` y asigna el resultado al registro `t`. Si queremos construir la máquina GCD sin usar una operación de resto

primitiva, debemos especificar cómo calcular los restos en términos de operaciones más simples, como la resta. De hecho, podemos escribir un procedimiento de Scheme que encuentre los restos de esta manera:

(define (remainder n d) (if (< n d) n (remainder (- n d) d)))

Figura 5.5: Rutas de datos y controlador para la máquina GCD elaborada.

Por lo tanto, podemos reemplazar la operación de resto en las rutas de datos de la máquina GCD con una operación de resta y una prueba de comparación. La Figura 5.5 muestra las rutas de datos y el controlador de la máquina elaborada. La instrucción

(assign t (op rem) (reg a) (reg b))

en la definición del controlador del GCD se reemplaza por una secuencia de instrucciones que contiene un bucle, como se muestra en la Figura 5.6.

Figura 5.6: ↓ Secuencia de instrucciones del controlador para la máquina GCD de la Figura 5.5.

(controller test-b (test (op =) (reg b) (const 0)) (branch (label gcd-done)) (assign t (reg a)) rem-loop (test (op <) (reg t) (reg b)) (branch (label rem-done)) (assign t (op -) (reg t) (reg b)) (goto (label rem-loop)) rem-done (assign a (reg b)) (assign b (reg t)) (goto (label test-b)) gcd-done)

Ejercicio 5.3: Diseñe una máquina para calcular raíces cuadradas usando el método de Newton, como se describe en Section 1.1.7:

(define (sqrt x) (define (good-enough? guess) (< (abs (- (square guess) x)) 0.001)) (define (improve guess) (average guess (/ x guess))) (define (sqrt-iter guess) (if (good-enough? guess) guess (sqrt-iter (improve guess)))) (sqrt-iter 1.0))

Comience asumiendo que las operaciones `good/enough?` e `improve` están disponibles como primitivas. Luego muestre cómo expandirlas en términos de operaciones aritméticas. Describa cada versión del diseño de la máquina `sqrt` dibujando un diagrama de ruta de datos y escribiendo una definición de controlador en el lenguaje de máquina de registros.

Subrutinas

Al diseñar una máquina para realizar un cálculo, a menudo preferiríamos

organizar los componentes para que sean compartidos por diferentes partes del cálculo en lugar de duplicar los componentes. Considere una máquina que incluye dos cálculos de MCD—uno que encuentra el MCD del contenido de los registros `a` y `b` y otro que encuentra el MCD del contenido de los registros `c` y `d`. Podríamos comenzar asumiendo que tenemos una operación primitiva `mcd`, y luego expandir las dos instancias de `mcd` en términos de operaciones más primitivas. La Figura 5.7 muestra solo las porciones de MCD de las rutas de datos de la máquina resultante, sin mostrar cómo se conectan con el resto de la máquina. La figura también muestra las porciones correspondientes de la secuencia del controlador de la máquina.

Figura 5.7: Porciones de las rutas de datos y la secuencia del controlador para una máquina con dos cálculos de MCD.

Esta máquina tiene dos cajas de operación de resto y dos cajas para probar la igualdad. Si los componentes duplicados son complicados, como lo es la caja de resto, esta no será una forma económica de construir la máquina. Podemos evitar duplicar los componentes de la ruta de datos usando los mismos componentes para ambos cálculos de MCD, siempre que hacerlo no afecte al resto del cálculo de la máquina más grande. Si los valores en los registros `a` y `b` no son necesarios para cuando el controlador llega a `gcd/2` (o si estos valores pueden moverse a otros registros para su custodia), podemos cambiar la máquina para que use los registros `a` y `b`, en lugar de los registros `c` y `d`, al calcular el segundo MCD así como el primero. Si hacemos esto, obtenemos la secuencia del controlador que se muestra en la Figura 5.8.

Figura 5.8: ↓ Porciones de la secuencia del controlador para una máquina que usa los mismos componentes de la ruta de datos para dos cálculos de MCD diferentes.

gcd-1 (test (op =) (reg b) (const 0)) (branch (label after-gcd-1)) (assign t (op rem) (reg a) (reg b)) (assign a (reg b)) (assign b (reg t)) (goto (label gcd-1)) after-gcd-1 … gcd-2 (test (op =) (reg b) (const 0)) (branch (label after-gcd-2)) (assign t (op rem) (reg a) (reg b)) (assign a (reg b)) (assign b (reg t)) (goto (label gcd-2)) after-gcd-2

Hemos eliminado los componentes duplicados de la ruta de datos (de modo que las rutas de datos vuelven a ser como en la Figura 5.1), pero el controlador ahora tiene dos secuencias de MCD que solo difieren en sus etiquetas de punto de entrada. Sería mejor reemplazar estas dos secuencias por bifurcaciones a una sola secuencia—una `mcd` *subrutina*—al final de la cual bifurcamos de vuelta al lugar correcto en la secuencia de instrucciones

principal. Podemos lograr esto de la siguiente manera: Antes de bifurcar a `mcd`, colocamos un valor distintivo (como 0 o 1) en un registro especial, `continuar`. Al final de la subrutina `mcd`, volvemos ya sea a `después/mcd/1` o a `después/mcd/2`, dependiendo del valor del registro `continuar`. La Figura 5.9 muestra la porción relevante de la secuencia del controlador resultante, que incluye solo una copia de las instrucciones de `mcd`.

Figura 5.9: ↓ Usando un registro `continue` para evitar la secuencia de controlador duplicada en la Figura 5.8.

```
gcd (test (op =) (reg b) (const 0)) (branch (label gcd-done)) (assign t (op
rem) (reg a) (reg b)) (assign a (reg b)) (assign b (reg t)) (goto (label gcd))
gcd-done (test (op =) (reg continue) (const 0)) (branch (label after-
gcd-1)) (goto (label after-gcd-2)) …  ;; Antes de bifurcar a gcd desde el
primer lugar donde  ;; se necesita, colocamos 0 en el registro continue
(assign continue (const 0)) (goto (label gcd)) after-gcd-1  …

 ;; Antes del segundo uso de gcd, colocamos 1  ;; en el registro
continue  (assign continue (const 1)) (goto (label gcd)) after-gcd-2
```

Este es un enfoque razonable para manejar problemas pequeños, pero sería incómodo si hubiera muchas instancias de cálculos de MCD en la secuencia del controlador. Para decidir dónde continuar la ejecución después de la subrutina `gcd`, necesitaríamos pruebas en las rutas de datos e instrucciones de bifurcación en el controlador para todos los lugares que usan `gcd`. Un método más poderoso para implementar subrutinas es hacer que el registro `continue` contenga la etiqueta del punto de entrada en la secuencia del controlador en el que la ejecución debe continuar cuando la subrutina termina. Implementar esta estrategia requiere un nuevo tipo de conexión entre las rutas de datos y el controlador de una máquina de registros: debe haber una forma de asignar a un registro una etiqueta en la secuencia del controlador de tal manera que este valor pueda ser obtenido del registro y utilizado para continuar la ejecución en el punto de entrada designado.

Para reflejar esta capacidad, extenderemos la instrucción `assign` del lenguaje de máquina de registros para permitir que a un registro se le asigne como valor una etiqueta de la secuencia del controlador (como un tipo especial de constante). También extenderemos la instrucción `goto` para permitir que la ejecución continúe en el punto de entrada descrito por el contenido de un registro en lugar de solo en un punto de entrada descrito por una etiqueta constante. Usando estas nuevas construcciones podemos terminar la subrutina `gcd` con una bifurcación a la ubicación almacenada en el registro `continue`. Esto conduce a la secuencia del controlador que se muestra en la Figura 5.10.

Figura 5.10: ↓ Asignar etiquetas al registro `continue` simplifica y generaliza la estrategia mostrada en la Figura 5.9.

```
gcd (test (op =) (reg b) (const 0)) (branch (label gcd-done)) (assign t (op rem) (reg a) (reg b)) (assign a (reg b)) (assign b (reg t)) (goto (label gcd)) gcd-done (goto (reg continue)) … ;; Antes de llamar a gcd, asignamos a continue ;; la etiqueta a la que gcd debe regresar. (assign continue (label after-gcd-1)) (goto (label gcd)) after-gcd-1 … ;; Aquí está la segunda llamada a gcd, ;; con una continuación diferente. (assign continue (label after-gcd-2)) (goto (label gcd)) after-gcd-2
```

Una máquina con más de una subrutina podría usar múltiples registros de continuación (p. ej., `gcd/continue`, `factorial/continue`) o podríamos hacer que todas las subrutinas compartan un único registro `continue`. Compartir es más económico, pero debemos tener cuidado si tenemos una subrutina (`sub1`) que llama a otra subrutina (`sub2`). A menos que `sub1` guarde el contenido de `continue` en algún otro registro antes de configurar `continue` para la llamada a `sub2`, `sub1` no sabrá a dónde ir cuando termine. El mecanismo desarrollado en la siguiente sección para manejar la recursión también proporciona una mejor solución a este problema de llamadas a subrutinas anidadas.

Usando una Pila para Implementar la Recursión

Con las ideas ilustradas hasta ahora, podemos implementar cualquier proceso iterativo especificando una máquina de registros que tenga un registro correspondiente a cada variable de estado del proceso. La máquina ejecuta repetidamente un bucle de controlador, cambiando el contenido de los registros, hasta que se satisface alguna condición de terminación. En cada punto de la secuencia del controlador, el estado de la máquina (que representa el estado del proceso iterativo) está completamente determinado por el contenido de los registros (los valores de las variables de estado).

Sin embargo, la implementación de procesos recursivos requiere un mecanismo adicional. Considere el siguiente método recursivo para calcular factoriales, que examinamos por primera vez en Section 1.2.1:

```
(define (factorial n) (if (= n 1) 1 (* (factorial (- n 1)) n)))
```

Como vemos en el procedimiento, calcular $n!$ requiere calcular $(n - 1)!$. Nuestra máquina MCD, modelada en el procedimiento

```
(define (gcd a b) (if (= b 0) a (gcd b (remainder a b))))
```

de manera similar tenía que calcular otro MCD. Pero hay una diferencia importante entre el procedimiento `mcd`, que reduce el cálculo original a un nuevo cálculo de MCD, y `factorial`, que requiere calcular otro factorial como un subproblema. En el MCD, la respuesta al nuevo cálculo de MCD es la respuesta al problema original. Para calcular el siguiente MCD, simplemente colocamos los nuevos argumentos en los registros de entrada de la máquina de MCD y reutilizamos las rutas de datos de la máquina ejecutando la misma secuencia de controlador. Cuando la máquina termina de resolver el problema final del MCD, ha completado todo el cálculo.

En el caso del factorial (o cualquier proceso recursivo) la respuesta al nuevo subproblema de factorial no es la respuesta al problema original. El valor obtenido para $(n - 1)!$ debe multiplicarse por n para obtener la respuesta final. Si intentamos imitar el diseño del MCD, y resolvemos el subproblema de factorial decrementando el registro `n` y volviendo a ejecutar la máquina de factorial, ya no tendremos disponible el valor antiguo de `n` por el cual multiplicar el resultado. Por lo tanto, necesitamos una segunda máquina de factorial para trabajar en el subproblema. Este segundo cálculo de factorial a su vez tiene un subproblema de factorial, que requiere una tercera máquina de factorial, y así sucesivamente. Dado que cada máquina de factorial contiene otra máquina de factorial dentro de ella, la máquina total contiene un nido infinito de máquinas similares y, por lo tanto, no puede construirse a partir de un número fijo y finito de partes.

Sin embargo, podemos implementar el proceso factorial como una máquina de registros si podemos hacer arreglos para usar los mismos componentes para cada instancia anidada de la máquina. Específicamente, la máquina que calcula $n!$ debería usar los mismos componentes para trabajar en el subproblema de calcular $(n - 1)!$, en el subproblema para $(n - 2)!$, y así sucesivamente. Esto es plausible porque, aunque el proceso factorial dicta que se necesita un número ilimitado de copias de la misma máquina para realizar un cálculo, solo una de estas copias necesita estar activa en un momento dado. Cuando la máquina encuentra un subproblema recursivo, puede suspender el trabajo en el problema principal, reutilizar las mismas partes físicas para trabajar en el subproblema, y luego continuar la computación suspendida.

En el subproblema, el contenido de los registros será diferente al que tenían en el problema principal. (En este caso, el registro `n` se decrementa). Para poder continuar la computación suspendida, la máquina debe guardar el contenido de cualquier registro que se necesite después de que se resuelva el subproblema para que estos puedan ser restaurados para continuar la computación suspendida. En el caso del factorial, guardaremos el valor antiguo de `n`, para restaurarlo cuando hayamos terminado de calcular el

factorial del registro `n` decrementado.[2]

Dado que no hay un límite *a priori* en la profundidad de las llamadas recursivas anidadas, es posible que necesitemos guardar un número arbitrario de valores de registro. Estos valores deben restaurarse en el orden inverso en que se guardaron, ya que en un nido de recursiones el último subproblema en entrar es el primero en terminar. Esto dicta el uso de una *pila*, o estructura de datos "último en entrar, primero en salir", para guardar los valores de los registros. Podemos extender el lenguaje de máquina de registros para incluir una pila agregando dos tipos de instrucciones: los valores se colocan en la pila usando una instrucción `save` y se restauran de la pila usando una instrucción `restore`. Después de que una secuencia de valores ha sido `guardada` en la pila, una secuencia de `restauraciones` recuperará estos valores en orden inverso.[3]

Con la ayuda de la pila, podemos reutilizar una sola copia de las rutas de datos de la máquina factorial para cada subproblema factorial. Hay un problema de diseño similar en la reutilización de la secuencia del controlador que opera las rutas de datos. Para volver a ejecutar el cálculo factorial, el controlador no puede simplemente volver al principio, como con un proceso iterativo, porque después de resolver el subproblema de $(n - 1)!$ la máquina aún debe multiplicar el resultado por n. El controlador debe suspender su cálculo de $n!$, resolver el subproblema de $(n - 1)!$, luego continuar su cálculo de $n!$. Esta visión del cálculo factorial sugiere el uso del mecanismo de subrutina descrito en Section 5.1.3, que hace que el controlador use un registro `continue` para transferirse a la parte de la secuencia que resuelve un subproblema y luego continúe donde lo dejó en el problema principal. Por lo tanto, podemos hacer una subrutina factorial que regrese al punto de entrada almacenado en el registro `continue`. Alrededor de cada llamada a la subrutina, guardamos y restauramos `continue` al igual que lo hacemos con el registro `n`, ya que cada "nivel" del cálculo factorial usará el mismo registro `continue`. Es decir, la subrutina factorial debe poner un nuevo valor en `continue` cuando se llama a sí misma para un subproblema, pero necesitará el valor antiguo para regresar al lugar que la llamó para resolver un subproblema.

Figura 5.11: Una máquina factorial recursiva.

Aunque en principio el cálculo del factorial requiere una máquina infinita, la máquina de la Figura 5.11 es en realidad finita excepto por la pila, que es potencialmente ilimitada. Sin embargo, cualquier implementación física particular de una pila será de tamaño finito, y esto limitará la profundidad de

las llamadas recursivas que la máquina puede manejar. Esta implementación del factorial ilustra la estrategia general para realizar algoritmos recursivos como máquinas de registro ordinarias aumentadas con pilas. Cuando se encuentra un subproblema recursivo, guardamos en la pila los registros cuyos valores actuales serán necesarios después de que se resuelva el subproblema, resolvemos el subproblema recursivo, luego restauramos los registros guardados y continuamos la ejecución en el problema principal. El registro `continue` siempre debe guardarse. Si hay otros registros que deben guardarse depende de la máquina en particular, ya que no todos los cálculos recursivos necesitan los valores originales de los registros que se modifican durante la solución del subproblema (ver Ejercicio 5.4).

Una doble recursión

Examinemos un proceso recursivo más complejo, el cálculo árbol-recursivo de los números de Fibonacci, que introdujimos en Section 1.2.2:

(define (fib n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))

Al igual que con el factorial, podemos implementar el cálculo recursivo de Fibonacci como una máquina de registros con los registros `n`, `val` y `continue`. La máquina es más compleja que la del factorial, porque hay dos lugares en la secuencia del controlador donde necesitamos realizar llamadas recursivas: una para calcular ${\rm Fib}(n - 1)$ y otra para calcular ${\rm Fib}(n - 2)$. Para preparar cada una de estas llamadas, guardamos los registros cuyos valores serán necesarios más tarde, establecemos el registro `n` al número cuyo Fib necesitamos calcular recursivamente ($n - 1$ o $n - 2$), y asignamos a `continue` el punto de entrada en la secuencia principal al que se debe regresar (`afterfib/n/1` o `afterfib/n/2`, respectivamente). Luego vamos a `fib/loop`. Cuando regresamos de la llamada recursiva, la respuesta está en `val`. La Figura 5.12 muestra la secuencia del controlador para esta máquina.

Figura 5.12: ↓ Controlador para una máquina para calcular números de Fibonacci.

(controller (assign continue (label fib-done)) fib-loop (test (op <) (reg n) (const 2)) (branch (label immediate-answer)) ;; set up to compute Fib($n - 1$) (save continue) (assign continue (label afterfib-n-1)) (save n) ; save old value of `n` (assign n (op -) (reg n) (const 1)) ; clobber `n` to `n/1` (goto (label fib-loop)) ; perform recursive call afterfib-n-1 ; upon return, `val` contains Fib($n - 1$) (restore n) (restore continue) ;; set up to compute Fib($n - 2$) (assign n (op -) (reg n) (const 2)) (save continue) (assign continue (label afterfib-n-2)) (save val) ; save Fib($n - 1$) (goto

(label fib-loop)) afterfib-n-2 ; upon return, `val` contains Fib($n - 2$) (assign n (reg val)) ; `n` now contains Fib($n - 2$) (restore val) ; `val` now contains Fib($n - 1$) (restore continue) (assign val ; Fib($n - 1$) + Fib($n - 2$) (op +) (reg val) (reg n)) (goto (reg continue)) ; return to caller, answer is in `val` immediate-answer (assign val (reg n)) ; base case: Fib(n) = n (goto (reg continue)) fib-done)

Ejercicio 5.4: Especifique máquinas de registros que implementen cada uno de los siguientes procedimientos. Para cada máquina, escriba una secuencia de instrucciones del controlador y dibuje un diagrama que muestre las rutas de datos.

1. Exponenciación recursiva:

   (define (expt b n) (if (= n 0) 1 (* b (expt b (- n 1)))))

2. Exponenciación iterativa:

   (define (expt b n) (define (expt-iter counter product) (if (= counter 0) product (expt-iter (- counter 1) (* b product)))) (expt-iter n 1))

Ejercicio 5.5: Simule a mano las máquinas de factorial y Fibonacci, usando alguna entrada no trivial (que requiera la ejecución de al menos una llamada recursiva). Muestre el contenido de la pila en cada punto significativo de la ejecución.

Ejercicio 5.6: Ben Bitdiddle observa que la secuencia del controlador de la máquina de Fibonacci tiene un `save` extra y un `restore` extra, que pueden eliminarse para hacer una máquina más rápida. ¿Dónde están estas instrucciones?

Resumen de Instrucciones

Una instrucción del controlador en nuestro lenguaje de máquina de registros tiene una de las siguientes formas, donde cada ⟨*entradai*⟩ es ya sea `(reg` ⟨*nombre/de/registro*⟩`)` o `(const` ⟨*valor/constante*⟩`)`. Estas instrucciones fueron introducidas en Section 5.1.1:

(assign < *nombre-de-registro* > (reg < *nombre-de-registro* >)) (assign < *nombre-de-registro* > (const < *valor-constante* >)) (assign < *nombre-de-registro* > (op < *nombre-de-operación* >) < *entrada* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$ … < *entrada* $\color{SchemeDark}_{\hbox{\ttfamily\itshape\scriptsize n}}\rangle$) (perform (op < *nombre-de-operación* >) < *entrada* $\color{SchemeDark}

_{\hbox{\ttfamily\scriptsize 1}}\rangle$ … ⟨ *entrada* $ \color{SchemeDark}_{\hbox{\ttfamily\itshape\scriptsize n}}\rangle$) (test (op ⟨ *nombre-de-operación* ⟩) ⟨ *entrada* $\color{SchemeDark} _{\hbox{\ttfamily\scriptsize 1}}\rangle$ … ⟨ *entrada* $ \color{SchemeDark}_{\hbox{\ttfamily\itshape\scriptsize n}}\rangle$) (branch (label ⟨ *nombre-de-etiqueta* ⟩)) (goto (label ⟨ *nombre-de-etiqueta* ⟩))

El uso de registros para guardar etiquetas se introdujo en Section 5.1.3:

(assign ⟨ *register-name* ⟩ (label ⟨ *label-name* ⟩)) (goto (reg ⟨ *register-name* ⟩))

Las instrucciones para usar la pila se introdujeron en Section 5.1.4:

(save ⟨ *nombre-de-registro* ⟩) (restore ⟨ *nombre-de-registro* ⟩)

El único tipo de ⟨*valor-constante*⟩ que hemos visto hasta ahora es un número, pero más adelante usaremos cadenas, símbolos y listas. Por ejemplo,

(const "abc") es la cadena "abc", (const abc) es el símbolo abc, (const (a b c)) es la lista (a b c), y (const ()) es la lista vacía.

Un Simulador de Máquina de Registros

Para obtener una buena comprensión del diseño de las máquinas de registros, debemos probar las máquinas que diseñamos para ver si funcionan como se espera. Una forma de probar un diseño es simular a mano la operación del controlador, como en el Ejercicio 5.5. Pero esto es extremadamente tedioso para todas las máquinas, excepto las más simples. En esta sección construimos un simulador para las máquinas descritas en el lenguaje de la máquina de registros. El simulador es un programa de Scheme con cuatro procedimientos de interfaz. El primero utiliza una descripción de una máquina de registros para construir un modelo de la máquina (una estructura de datos cuyas partes corresponden a las partes de la máquina que se va a simular), y los otros tres nos permiten simular la máquina manipulando el modelo:

> (make-machine ⟨ *nombres-de-registros* ⟩ ⟨ *operaciones* ⟩ ⟨ *controlador* ⟩)
>
> construye y devuelve un modelo de la máquina con los registros, operaciones y controlador dados.

(set-register-contents! $\color{SchemeDark}\langle\kern0.08em$ *modelo-de-máquina* $\color{SchemeDark}\rangle$ $\color{SchemeDark}\langle$ *nombre-de-registro* $\color{SchemeDark}\rangle$ $\color{SchemeDark}\langle$ *valor* $\color{SchemeDark}\rangle$)

almacena un valor en un registro simulado en la máquina dada.

(get-register-contents $\color{SchemeDark}\langle\kern0.08em$ *modelo-de-máquina* $\color{SchemeDark}\rangle$ $\color{SchemeDark}\langle$ *nombre-de-registro* $\color{SchemeDark}\rangle$)

devuelve el contenido de un registro simulado en la máquina dada.

(start $\color{SchemeDark}\langle\kern0.08em$ *machine-model* $\color{SchemeDark}\rangle$)

simula la ejecución de la máquina dada, comenzando desde el principio de la secuencia del controlador y deteniéndose cuando llega al final de la secuencia.

Como ejemplo de cómo se usan estos procedimientos, podemos definir `gcd/machine` como un modelo de la máquina MCD de Section 5.1.1 de la siguiente manera:

(define gcd-machine (make-machine '(a b t) (list (list 'rem remainder) (list '= =)) '(test-b (test (op =) (reg b) (const 0)) (branch (label gcd-done)) (assign t (op rem) (reg a) (reg b)) (assign a (reg b)) (assign b (reg t)) (goto (label test-b)) gcd-done)))

El primer argumento de `make/machine` es una lista de nombres de registros. El siguiente argumento es una tabla (una lista de listas de dos elementos) que empareja cada nombre de operación con un procedimiento de Scheme que implementa la operación (es decir, produce el mismo valor de salida dado los mismos valores de entrada). El último argumento especifica el controlador como una lista de etiquetas e instrucciones de máquina, como en Section 5.1.

Para calcular los MCD con esta máquina, establecemos los registros de entrada, iniciamos la máquina y examinamos el resultado cuando la simulación termina:

(set-register-contents! gcd-machine 'a 206) *done* (set-register-contents! gcd-machine 'b 40) *done* (start gcd-machine) *done* (get-register-contents gcd-machine 'a) *2*

Este cálculo se ejecutará mucho más lentamente que un procedimiento `mcd` escrito en Scheme, porque simularemos instrucciones de máquina de bajo nivel, como `assign`, con operaciones mucho más complejas.

Ejercicio 5.7: Utilice el simulador para probar las máquinas que diseñó

en el Ejercicio 5.4.

El Modelo de la Máquina

El modelo de máquina generado por `make/machine` se representa como un procedimiento con estado local utilizando las técnicas de paso de mensajes desarrolladas en el Capítulo 3. Para construir este modelo, `make/machine` comienza llamando al procedimiento `make/new/machine` para construir las partes del modelo de máquina que son comunes a todas las máquinas de registros. Este modelo de máquina básico construido por `make/new/machine` es esencialmente un contenedor para algunos registros y una pila, junto con un mecanismo de ejecución que procesa las instrucciones del controlador una por una.

`make/machine` luego extiende este modelo básico (enviándole mensajes) para incluir los registros, operaciones y controlador de la máquina particular que se está definiendo. Primero asigna un registro en la nueva máquina para cada uno de los nombres de registro suministrados e instala las operaciones designadas en la máquina. Luego usa un *ensamblador* (descrito a continuación en Section 5.2.2) para transformar la lista del controlador en instrucciones para la nueva máquina e instala estas como la secuencia de instrucciones de la máquina. `make/machine` devuelve como su valor el modelo de máquina modificado.

(define (make-machine register-names ops controller-text) (let ((machine (make-new-machine))) (for-each (lambda (register-name) ((machine 'allocate-register) register-name)) register-names) ((machine 'install-operations) ops) ((machine 'install-instruction-sequence) (assemble controller-text machine)) machine))

Registros

Representaremos un registro como un procedimiento con estado local, como en el Capítulo 3. El procedimiento `make/register` crea un registro que contiene un valor que puede ser accedido o cambiado:

(define (make-register name) (let ((contents '*unassigned*)) (define (dispatch message) (cond ((eq? message 'get) contents) ((eq? message 'set) (lambda (value) (set! contents value))) (else (error "Unknown request: REGISTER" message)))) dispatch))

Los siguientes procedimientos se utilizan para acceder a los registros:

(define (get-contents register) (register 'get)) (define (set-contents! register value) ((register 'set) value))

La pila

También podemos representar una pila como un procedimiento con estado local. El procedimiento `make/stack` crea una pila cuyo estado local consiste en una lista de los elementos de la pila. Una pila acepta solicitudes para `push` un elemento en la pila, para `pop` el elemento superior de la pila y devolverlo, y para `initialize` la pila a vacía.

(define (make-stack) (let ((s '())) (define (push x) (set! s (cons x s))) (define (pop) (if (null? s) (error "Empty stack: POP") (let ((top (car s))) (set! s (cdr s)) top))) (define (initialize) (set! s '()) 'done) (define (dispatch message) (cond ((eq? message 'push) push) ((eq? message 'pop) (pop)) ((eq? message 'initialize) (initialize)) (else (error "Unknown request: STACK" message)))) dispatch))

Los siguientes procedimientos se utilizan para acceder a las pilas:

(define (pop stack) (stack 'pop)) (define (push stack value) ((stack 'push) value))

La máquina básica

El procedimiento `make/new/machine`, que se muestra en la Figura 5.13, construye un objeto cuyo estado local consiste en una pila, una secuencia de instrucciones inicialmente vacía, una lista de operaciones que inicialmente contiene una operación para inicializar la pila, y una *tabla de registros* que inicialmente contiene dos registros, llamados `flag` y `pc` (para "contador de programa"). El procedimiento interno `allocate/register` agrega nuevas entradas a la tabla de registros, y el procedimiento interno `lookup/register` busca registros en la tabla.

El registro `flag` se utiliza para controlar la bifurcación en la máquina simulada. Las instrucciones `test` establecen el contenido de `flag` en el resultado de la prueba (verdadero o falso). Las instrucciones `branch` deciden si bifurcar o no examinando el contenido de `flag`.

El registro `pc` determina la secuencia de instrucciones a medida que la máquina se ejecuta. Esta secuencia es implementada por el procedimiento interno `execute`. En el modelo de simulación, cada instrucción de la máquina es una estructura de datos que incluye un procedimiento sin argumentos, llamado el *procedimiento de ejecución de la instrucción*, tal que llamar a este

procedimiento simula la ejecución de la instrucción. A medida que la simulación se ejecuta, `pc` apunta al lugar en la secuencia de instrucciones que comienza con la siguiente instrucción a ejecutar. `execute` obtiene esa instrucción, la ejecuta llamando al procedimiento de ejecución de la instrucción, y repite este ciclo hasta que no haya más instrucciones que ejecutar (es decir, hasta que `pc` apunte al final de la secuencia de instrucciones).

Figura 5.13: ↓ El procedimiento `make/new/machine`, que implementa el modelo básico de la máquina.

```
(define (make-new-machine)
  (let ((pc (make-register 'pc))
        (flag (make-register 'flag))
        (stack (make-stack))
        (the-instruction-sequence '()))
    (let ((the-ops
           (list (list 'initialize-stack
                       (lambda () (stack 'initialize)))))
          (register-table
           (list (list 'pc pc) (list 'flag flag))))
      (define (allocate-register name)
        (if (assoc name register-table)
            (error "Multiply defined register: "
                   name)
            (set! register-table
                  (cons (list name (make-register name))
                        register-table)))
        'register-allocated)
      (define (lookup-register name)
        (let ((val (assoc name register-table)))
          (if val
              (cadr val)
              (error "Unknown register:"
                     name))))
      (define (execute)
        (let ((insts (get-contents pc)))
          (if (null? insts)
              'done
              (begin
                ((instruction-execution-proc
                  (car insts)))
                (execute)))))
      (define (dispatch message)
        (cond ((eq? message 'start)
               (set-contents! pc
                              the-instruction-sequence)
               (execute))
              ((eq? message 'install-instruction-
                    sequence)
               (lambda (seq)
                 (set! the-instruction-sequence seq)))
              ((eq? message 'allocate-register)
               allocate-register)
              ((eq? message 'get-register)
               lookup-register)
              ((eq? message 'install-operations)
               (lambda (ops)
                 (set! the-ops
                       (append the-ops ops))))
              ((eq? message 'stack) stack)
              ((eq? message 'operations)
               the-ops)
              (else
               (error "Unknown request: MACHINE"
                      message))))
      dispatch)))
```

Como parte de su operación, cada procedimiento de ejecución de instrucción modifica `pc` para indicar la siguiente instrucción a ejecutar. Las instrucciones `branch` y `goto` cambian `pc` para que apunte al nuevo destino. Todas las demás instrucciones simplemente avanzan `pc`, haciéndolo apuntar a la siguiente instrucción de la secuencia. Observe que cada llamada a `execute` llama a `execute` de nuevo, pero esto no produce un bucle infinito porque ejecutar el procedimiento de ejecución de la instrucción cambia el contenido de `pc`.

`make/new/machine` devuelve un procedimiento `dispatch` que implementa el acceso por paso de mensajes al estado interno. Nótese que el inicio de la máquina se logra estableciendo `pc` al principio de la secuencia de instrucciones y llamando a `execute`.

Por conveniencia, proporcionamos una interfaz procedimental alternativa a la operación `start` de una máquina, así como procedimientos para establecer y examinar el contenido de los registros, como se especificó al principio de Section 5.2:

(define (start machine) (machine 'start)) (define (get-register-contents machine register-name) (get-contents (get-register machine register-name))) (define (set-register-contents! machine register-name value) (set-contents! (get-register machine register-name) value) 'done)

Estos procedimientos (y muchos procedimientos en Section 5.2.2 y Section 5.2.3) usan lo siguiente para buscar el registro con un nombre dado en una máquina dada:

(define (get-register machine reg-name) ((machine 'get-register) reg-name))

El Ensamblador

El ensamblador transforma la secuencia de expresiones del controlador para una máquina en una lista correspondiente de instrucciones de máquina, cada una con su procedimiento de ejecución. En general, el ensamblador se parece mucho a los evaluadores que estudiamos en el Capítulo 4—hay un lenguaje de entrada (en este caso, el lenguaje de máquina de registros) y debemos realizar una acción apropiada para cada tipo de expresión en el lenguaje.

La técnica de producir un procedimiento de ejecución para cada instrucción es justamente la que usamos en Section 4.1.7 para acelerar el evaluador separando el análisis de la ejecución en tiempo de ejecución. Como vimos en el Capítulo 4, se podía realizar mucho análisis útil de las expresiones de Scheme sin conocer los valores reales de las variables. Aquí, análogamente, se puede realizar mucho análisis útil de las expresiones del lenguaje de máquina de registros sin conocer el contenido real de los registros de la máquina. Por ejemplo, podemos reemplazar las referencias a registros por punteros a los objetos de registro, y podemos reemplazar las referencias a etiquetas por punteros al lugar en la secuencia de instrucciones que la etiqueta designa.

Antes de que pueda generar los procedimientos de ejecución de instrucciones, el ensamblador debe saber a qué se refieren todas las etiquetas, por lo que comienza escaneando el texto del controlador para separar las etiquetas de las instrucciones. A medida que escanea el texto, construye tanto una lista de instrucciones como una tabla que asocia cada etiqueta con un puntero a esa lista. Luego, el ensamblador aumenta la lista de instrucciones insertando el procedimiento de ejecución para cada instrucción.

El procedimiento `assemble` es la entrada principal al ensamblador. Toma el texto del controlador y el modelo de la máquina como argumentos y devuelve la secuencia de instrucciones que se almacenará en el modelo. `assemble` llama a `extract/labels` para construir la lista de instrucciones inicial y la tabla de etiquetas a partir del texto del controlador proporcionado. El segundo argumento de `extract/labels` es un procedimiento que se llamará para procesar estos resultados: Este procedimiento utiliza `update/insts!` para generar los procedimientos de ejecución de instrucciones e insertarlos en la lista de instrucciones, y devuelve la lista modificada.

(define (assemble controller-text machine) (extract-labels controller-text (lambda (insts labels) (update-insts! insts labels machine) insts)))

`extract/labels` toma como argumentos una lista `text` (la secuencia de expresiones de instrucción del controlador) y un procedimiento `receive`. `receive` será llamado con dos valores: (1) una lista `insts` de estructuras de datos de instrucción, cada una conteniendo una instrucción de `text`; y (2) una tabla llamada `labels`, que asocia cada etiqueta de `text` con la posición en la lista `insts` que la etiqueta designa.

(define (extract-labels text receive) (if (null? text) (receive '() '()) (extract-labels (cdr text) (lambda (insts labels) (let ((next-inst (car text))) (if (symbol? next-inst) (receive insts (cons (make-label-entry next-inst insts) labels)) (receive (cons (make-instruction next-inst) insts) labels)))))))

`extract/labels` funciona escaneando secuencialmente los elementos del `text` y acumulando los `insts` y los `labels`. Si un elemento es un símbolo (y por lo tanto una etiqueta) se añade una entrada apropiada a la tabla `labels`. De lo contrario, el elemento se acumula en la lista `insts`.[4]

`update/insts!` modifica la lista de instrucciones, que inicialmente contiene solo el texto de las instrucciones, para incluir los procedimientos de ejecución correspondientes:

(define (update-insts! insts labels machine) (let ((pc (get-register machine 'pc)) (flag (get-register machine 'flag)) (stack (machine 'stack)) (ops (machine 'operations))) (for-each (lambda (inst) (set-instruction-execution-proc! inst (make-execution-procedure (instruction-text inst) labels machine pc flag stack ops))) insts)))

La estructura de datos de la instrucción de la máquina simplemente empareja el texto de la instrucción con el procedimiento de ejecución correspondiente. El procedimiento de ejecución no está todavía disponible cuando `extract/labels` construye la instrucción, y se inserta más tarde por `update/insts!`.

(define (make-instruction text) (cons text '())) (define (instruction-text inst) (car inst)) (define (instruction-execution-proc inst) (cdr inst)) (define (set-instruction-execution-proc! inst proc) (set-cdr! inst proc))

El texto de la instrucción no es utilizado por nuestro simulador, pero es útil tenerlo a mano para la depuración (ver Ejercicio 5.16).

Los elementos de la tabla de etiquetas son pares:

(define (make-label-entry label-name insts) (cons label-name insts))

Las entradas se buscarán en la tabla con

(define (lookup-label labels label-name) (let ((val (assoc label-name labels))) (if val (cdr val) (error "Undefined label: ASSEMBLE" label-name))))

> **Ejercicio 5.8:** El siguiente código de máquina de registros es ambiguo, porque la etiqueta `here` se define más de una vez:
>
> start (goto (label here)) here (assign a (const 3)) (goto (label there)) here (assign a (const 4)) (goto (label there)) there
>
> Con el simulador tal como está escrito, ¿cuál será el contenido del registro `a` cuando el control llegue a `there`? Modifique el procedimiento `extract/labels` para que el ensamblador señale un error si se utiliza el mismo nombre de etiqueta para indicar dos ubicaciones diferentes.

Generando Procedimientos de Ejecución* para Instrucciones

El ensamblador llama a `make/execution/procedure` para generar el procedimiento de ejecución para una instrucción. Al igual que el procedimiento `analyze` en el evaluador de Section 4.1.7, este despacha según el tipo de instrucción para generar el procedimiento de ejecución apropiado.

(define (make-execution-procedure inst labels machine pc flag stack ops) (cond ((eq? (car inst) 'assign) (make-assign inst machine labels ops pc)) ((eq? (car inst) 'test) (make-test inst machine labels ops flag pc)) ((eq? (car inst) 'branch) (make-branch inst machine labels flag pc)) ((eq? (car inst) 'goto) (make-goto inst machine labels pc)) ((eq? (car inst) 'save) (make-save inst machine stack pc)) ((eq? (car inst) 'restore) (make-restore inst machine stack pc)) ((eq? (car inst) 'perform) (make-perform inst machine labels ops pc)) (else (error "Unknown instruction type: ASSEMBLE" inst))))

Para cada tipo de instrucción en el lenguaje de máquina de registros, hay un generador que construye un procedimiento de ejecución apropiado. Los detalles de estos procedimientos determinan tanto la sintaxis como el significado de las instrucciones individuales en el lenguaje de máquina de registros. Utilizamos la abstracción de datos para aislar la sintaxis detallada de las expresiones de la máquina de registros del mecanismo general de ejecución, como hicimos para los evaluadores en Section 4.1.2, utilizando procedimientos de sintaxis para extraer y clasificar las partes de una instrucción.

instrucciones `assign`

El procedimiento `make/assign` maneja las instrucciones `assign`:

(define (make-assign inst machine labels operations pc) (let ((target (get-register machine (assign-reg-name inst))) (value-exp (assign-value-exp inst))) (let ((value-proc (if (operation-exp? value-exp) (make-operation-exp value-exp machine labels operations) (make-primitive-exp (car value-exp) machine labels)))) (lambda () ; procedimiento de ejecución para `assign` (set-contents! target (value-proc)) (advance-pc pc)))))

`make/assign` extrae el nombre del registro de destino (el segundo elemento de la instrucción) y la expresión de valor (el resto de la lista que forma la instrucción) de la instrucción `assign` utilizando los selectores

(define (assign-reg-name assign-instruction) (cadr assign-instruction)) (define (assign-value-exp assign-instruction) (cddr assign-instruction))

El nombre del registro se busca con `get/register` para producir el objeto de registro de destino. La expresión de valor se pasa a `make/operation/exp` si el valor es el resultado de una operación, y a `make/primitive/exp` en caso contrario. Estos procedimientos (mostrados a continuación) analizan la expresión de valor y producen un procedimiento de ejecución para el valor. Este es un procedimiento sin argumentos, llamado `value/proc`, que será evaluado durante la simulación para producir el valor real que se asignará al registro. Observe que el trabajo de buscar el nombre del registro y analizar la expresión de valor se realiza solo una vez, en tiempo de ensamblaje, no cada vez que se simula la instrucción. Este ahorro de trabajo es la razón por la que usamos procedimientos de ejecución, y corresponde directamente al ahorro de trabajo que obtuvimos al separar el análisis del programa de la ejecución en el evaluador de Section 4.1.7.

El resultado devuelto por `make/assign` es el procedimiento de ejecución para la instrucción `assign`. Cuando se llama a este procedimiento (por el

procedimiento `execute` del modelo de la máquina), establece el contenido del registro de destino al resultado obtenido al ejecutar `value/proc`. Luego avanza el `pc` a la siguiente instrucción ejecutando el procedimiento

(define (advance-pc pc) (set-contents! pc (cdr (get-contents pc))))

`advance/pc` es la terminación normal para todas las instrucciones excepto `branch` y `goto`.

instrucciones `test`, `branch`, y `goto`

`make/test` maneja las instrucciones `test` de manera similar. Extrae la expresión que especifica la condición a ser probada y genera un procedimiento de ejecución para ella. En tiempo de simulación, se llama al procedimiento para la condición, el resultado se asigna al registro `flag`, y se avanza el `pc`:

(define (make-test inst machine labels operations flag pc) (let ((condition (test-condition inst))) (if (operation-exp? condition) (let ((condition-proc (make-operation-exp condition machine labels operations))) (lambda () (set-contents! flag (condition-proc)) (advance-pc pc))) (error "Bad TEST instruction: ASSEMBLE" inst)))) (define (test-condition test-instruction) (cdr test-instruction))

El procedimiento de ejecución para una instrucción `branch` comprueba el contenido del registro `flag` y o bien establece el contenido del `pc` al destino del salto (si se toma el salto) o bien simplemente avanza el `pc` (si no se toma el salto). Observe que el destino indicado en una instrucción `branch` debe ser una etiqueta, y el procedimiento `make/branch` refuerza esto. Observe también que la etiqueta se busca en tiempo de ensamblaje, no cada vez que se simula la instrucción `branch`.

(define (make-branch inst machine labels flag pc) (let ((dest (branch-dest inst))) (if (label-exp? dest) (let ((insts (lookup-label labels (label-exp-label dest)))) (lambda () (if (get-contents flag) (set-contents! pc insts) (advance-pc pc)))) (error "Bad BRANCH instruction: ASSEMBLE" inst)))) (define (branch-dest branch-instruction) (cadr branch-instruction))

Una instrucción `goto` es similar a un salto, excepto que el destino puede especificarse ya sea como una etiqueta o como un registro, y no hay condición que comprobar—el `pc` siempre se establece en el nuevo destino.

(define (make-goto inst machine labels pc) (let ((dest (goto-dest inst))) (cond ((label-exp? dest) (let ((insts (lookup-label labels (label-exp-label dest))))

(lambda () (set-contents! pc insts)))) ((register-exp? dest) (let ((reg (get-register machine (register-exp-reg dest)))) (lambda () (set-contents! pc (get-contents reg))))) (else (error "Bad GOTO instruction: ASSEMBLE" inst))))) (define (goto-dest goto-instruction) (cadr goto-instruction))

Otras instrucciones

Las instrucciones de pila `save` y `restore` simplemente usan la pila con el registro designado y avanzan el `pc`:

(define (make-save inst machine stack pc) (let ((reg (get-register machine (stack-inst-reg-name inst)))) (lambda () (push stack (get-contents reg)) (advance-pc pc)))) (define (make-restore inst machine stack pc) (let ((reg (get-register machine (stack-inst-reg-name inst)))) (lambda () (set-contents! reg (pop stack)) (advance-pc pc)))) (define (stack-inst-reg-name stack-instruction) (cadr stack-instruction))

El tipo final de instrucción, manejado por `make/perform`, genera un procedimiento de ejecución para la acción a realizar. En tiempo de simulación, el procedimiento de acción se ejecuta y el `pc` avanza.

(define (make-perform inst machine labels operations pc) (let ((action (perform-action inst))) (if (operation-exp? action) (let ((action-proc (make-operation-exp action machine labels operations))) (lambda () (action-proc) (advance-pc pc))) (error "Bad PERFORM instruction: ASSEMBLE" inst)))) (define (perform-action inst) (cdr inst))

Procedimientos de ejecución para subexpresiones

El valor de una expresión `reg`, `label`, o `const` puede ser necesario para la asignación a un registro (`make/assign`) o como entrada para una operación (`make/operation/exp`, a continuación). El siguiente procedimiento genera procedimientos de ejecución para producir valores para estas expresiones durante la simulación:

(define (make-primitive-exp exp machine labels) (cond ((constant-exp? exp) (let ((c (constant-exp-value exp))) (lambda () c))) ((label-exp? exp) (let ((insts (lookup-label labels (label-exp-label exp)))) (lambda () insts))) ((register-exp? exp) (let ((r (get-register machine (register-exp-reg exp)))) (lambda () (get-contents r)))) (else (error "Unknown expression type: ASSEMBLE" exp))))

La sintaxis de las expresiones `reg`, `label` y `const` se determina por

(define (register-exp? exp) (tagged-list? exp 'reg)) (define (register-exp-reg

exp) (cadr exp)) (define (constant-exp? exp) (tagged-list? exp 'const)) (define (constant-exp-value exp) (cadr exp)) (define (label-exp? exp) (tagged-list? exp 'label)) (define (label-exp-label exp) (cadr exp))

Las instrucciones `assign`, `perform` y `test` pueden incluir la aplicación de una operación de la máquina (especificada por una expresión `op`) a algunos operandos (especificados por expresiones `reg` y `const`). El siguiente procedimiento produce un procedimiento de ejecución para una "expresión de operación" —una lista que contiene la operación y las expresiones de operando de la instrucción:

(define (make-operation-exp exp machine labels operations) (let ((op (lookup-prim (operation-exp-op exp) operations)) (aprocs (map (lambda (e) (make-primitive-exp e machine labels)) (operation-exp-operands exp)))) (lambda () (apply op (map (lambda (p) (p)) aprocs)))))

La sintaxis de las expresiones de operación está determinada por

(define (operation-exp? exp) (and (pair? exp) (tagged-list? (car exp) 'op))) (define (operation-exp-op operation-exp) (cadr (car operation-exp))) (define (operation-exp-operands operation-exp) (cdr operation-exp))

Observe que el tratamiento de las expresiones de operación es muy parecido al tratamiento de las aplicaciones de procedimientos por el procedimiento `analyze/application` en el evaluador de Section 4.1.7 en que generamos un procedimiento de ejecución para cada operando. En tiempo de simulación, llamamos a los procedimientos de operando y aplicamos el procedimiento de Scheme que simula la operación a los valores resultantes. El procedimiento de simulación se encuentra buscando el nombre de la operación en la tabla de operaciones de la máquina:

(define (lookup-prim symbol operations) (let ((val (assoc symbol operations))) (if val (cadr val) (error "Unknown operation: ASSEMBLE" symbol))))

> **Ejercicio 5.9:** El tratamiento de las operaciones de la máquina anterior permite que operen tanto con etiquetas como con constantes y contenidos de registros. Modifique los procedimientos de procesamiento de expresiones para hacer cumplir la condición de que las operaciones solo se pueden usar con registros y constantes.
>
> **Ejercicio 5.10:** Diseñe una nueva sintaxis para las instrucciones de la máquina de registros y modifique el simulador para usar su nueva sintaxis. ¿Puede implementar su nueva sintaxis sin cambiar ninguna parte del simulador excepto los procedimientos de sintaxis en esta sección?

Ejercicio 5.11: Cuando introdujimos `save` y `restore` en Section 5.1.4, no especificamos qué sucedería si intentara restaurar un registro que no fue el último guardado, como en la secuencia

(save y) (save x) (restore y)

Hay varias posibilidades razonables para el significado de `restore`:

1. `(restore y)` pone en `y` el último valor guardado en la pila, independientemente del registro del que provino ese valor. Esta es la forma en que nuestro simulador se comporta. Muestre cómo aprovechar este comportamiento para eliminar una instrucción de la máquina de Fibonacci de Section 5.1.4 (Figura 5.12).

2. `(restore y)` pone en `y` el último valor guardado en la pila, pero solo si ese valor fue guardado desde `y`; de lo contrario, señala un error. Modifique el simulador para que se comporte de esta manera. Tendrá que cambiar `save` para poner el nombre del registro en la pila junto con el valor.

3. `(restore y)` pone en `y` el último valor guardado desde `y` independientemente de qué otros registros se guardaron después de `y` y no se restauraron. Modifique el simulador para que se comporte de esta manera. Tendrá que asociar una pila separada con cada registro. Deberá hacer que la operación `initialize/stack` inicialice todas las pilas de registro.

Ejercicio 5.12: El simulador se puede utilizar para ayudar a determinar las rutas de datos necesarias para implementar una máquina con un controlador dado. Amplíe el ensamblador para almacenar la siguiente información en el modelo de la máquina:

- una lista de todas las instrucciones, sin duplicados, ordenadas por tipo de instrucción (`assign`, `goto`, etc.);

- una lista (sin duplicados) de los registros utilizados para mantener los puntos de entrada (estos son los registros a los que hacen referencia las instrucciones `goto`);

- una lista (sin duplicados) de los registros que se `save`d o `restore`d;

- para cada registro, una lista (sin duplicados) de las fuentes desde las que se asigna (por ejemplo, las fuentes para el registro `val` en la máquina factorial de Figura 5.11 son `(const 1)` y `((op *) (reg n) (reg val))`).

Amplíe la interfaz de paso de mensajes a la máquina para proporcionar acceso a esta nueva información. Para probar su analizador, defina la máquina de Fibonacci de Figura 5.12 y examine las listas que construyó.

Ejercicio 5.13: Modifique el simulador para que utilice la secuencia del controlador para determinar qué registros tiene la máquina en lugar de requerir una lista de registros como argumento para `make/machine`. En lugar de pre-asignar los registros en `make/machine`, puede asignarlos uno por uno cuando se ven por primera vez durante el ensamblaje de las instrucciones.

Monitoreo del Rendimiento de la Máquina

La simulación es útil no solo para verificar la corrección de un diseño de máquina propuesto, sino también para medir el rendimiento de la máquina. Por ejemplo, podemos instalar en nuestro programa de simulación un "medidor" que mida el número de operaciones de pila utilizadas en un cálculo. Para hacer esto, modificamos nuestra pila simulada para realizar un seguimiento del número de veces que se guardan los registros en la pila y la profundidad máxima alcanzada por la pila, y agregamos un mensaje a la interfaz de la pila que imprime las estadísticas, como se muestra a continuación. También agregamos una operación al modelo básico de la máquina para imprimir las estadísticas de la pila, inicializando `the/ops` en `make/new/machine` a

(list (list 'initialize-stack (lambda () (stack 'initialize))) (list 'print-stack-statistics (lambda () (stack 'print-statistics))))

Aquí está la nueva versión de `make/stack`:

(define (make-stack) (let ((s '()) (number-pushes 0) (max-depth 0) (current-depth 0)) (define (push x) (set! s (cons x s)) (set! number-pushes (+ 1 number-pushes)) (set! current-depth (+ 1 current-depth)) (set! max-depth (max current-depth max-depth))) (define (pop) (if (null? s) (error "Empty stack: POP") (let ((top (car s))) (set! s (cdr s)) (set! current-depth (- current-depth 1)) top))) (define (initialize) (set! s '()) (set! number-pushes 0) (set! max-depth 0) (set! current-depth 0) 'done) (define (print-statistics) (newline) (display (list 'total-pushes '= number-pushes 'maximum-depth '= max-depth))) (define (dispatch message) (cond ((eq? message 'push) push) ((eq? message 'pop) (pop)) ((eq? message 'initialize) (initialize)) ((eq? message 'print-statistics) (print-statistics)) (else (error "Unknown request: STACK" message)))) dispatch))

Del Ejercicio 5.15 al Ejercicio 5.19 se describen otras características útiles de

monitoreo y depuración que se pueden agregar al simulador de máquina de registros.

Ejercicio 5.14: Mida el número de pushes y la profundidad máxima de la pila requerida para calcular *n*! para varios valores pequeños de *n* usando la máquina factorial que se muestra en la Figura 5.11. A partir de sus datos, determine fórmulas en términos de *n* para el número total de operaciones de push y la profundidad máxima de la pila utilizada para calcular *n*! para cualquier $n > 1$. Tenga en cuenta que cada una de estas es una función lineal de *n* y, por lo tanto, está determinada por dos constantes. Para que se impriman las estadísticas, deberá aumentar la máquina factorial con instrucciones para inicializar la pila e imprimir las estadísticas. Es posible que también desee modificar la máquina para que lea repetidamente un valor para *n*, calcule el factorial e imprima el resultado (como hicimos para la máquina GCD en la Figura 5.4), de modo que no tenga que invocar repetidamente `get/register/contents`, `set/register/contents!` y `start`.

Ejercicio 5.15: Agregue el *conteo de instrucciones* a la simulación de la máquina de registros. Es decir, haga que el modelo de la máquina lleve un registro del número de instrucciones ejecutadas. Amplíe la interfaz del modelo de la máquina para aceptar un nuevo mensaje que imprima el valor del conteo de instrucciones y reinicie el conteo a cero.

Ejercicio 5.16: Amplíe el simulador para proporcionar un *seguimiento de instrucciones*. Es decir, antes de que se ejecute cada instrucción, el simulador debe imprimir el texto de la instrucción. Haga que el modelo de la máquina acepte mensajes `trace/on` y `trace/off` para activar y desactivar el seguimiento.

Ejercicio 5.17: Amplíe el seguimiento de instrucciones del Ejercicio 5.16 para que, antes de imprimir una instrucción, el simulador imprima cualquier etiqueta que preceda inmediatamente a esa instrucción en la secuencia del controlador. Tenga cuidado de hacer esto de una manera que no interfiera con el conteo de instrucciones (Ejercicio 5.15). Tendrá que hacer que el simulador retenga la información necesaria de la etiqueta.

Ejercicio 5.18: Modifique el procedimiento `make/register` de Section 5.2.1 para que los registros puedan ser rastreados. Los registros deben aceptar mensajes que activen y desactiven el rastreo. Cuando se rastrea un registro, la asignación de un valor al registro debe imprimir el nombre del registro, el contenido antiguo del registro y el nuevo contenido que se está asignando. Amplíe la interfaz al modelo de la máquina para

permitirle activar y desactivar el rastreo para los registros designados de la máquina.

Ejercicio 5.19: Alyssa P. Hacker quiere una característica de *punto de interrupción* en el simulador para ayudarla a depurar sus diseños de máquinas. Usted ha sido contratado para instalar esta característica para ella. Ella quiere poder especificar un lugar en la secuencia del controlador donde el simulador se detendrá y le permitirá examinar el estado de la máquina. Debe implementar un procedimiento

(set-breakpoint $\color{SchemeDark}\langle\kern0.08em$ *máquina* $\color{SchemeDark}\kern0.08em\rangle$ $\color{SchemeDark}\langle\kern0.08em$ *etiqueta* $\color{SchemeDark}\kern0.08em\rangle$ $\color{SchemeDark}\langle\kern0.08em$ *n* $\color{SchemeDark}\kern0.08em\rangle$)

que establece un punto de interrupción justo antes de la n^{th} instrucción después de la etiqueta dada. Por ejemplo,

(set-breakpoint gcd-machine 'test-b 4)

instala un punto de interrupción en `gcd/machine` justo antes de la asignación al registro `a`. Cuando el simulador llega al punto de interrupción, debe imprimir la etiqueta y el desplazamiento del punto de interrupción y dejar de ejecutar instrucciones. Alyssa puede entonces usar `get/register/contents` y `set/register/contents!` para manipular el estado de la máquina simulada. Luego debería poder continuar la ejecución diciendo

(proceed-machine $\color{SchemeDark}\langle\kern0.08em$ $ *máquina* $\color{SchemeDark}\kern0.08em\rangle$)

También debería poder eliminar un punto de interrupción específico mediante

(cancel-breakpoint $\color{SchemeDark}\langle\kern0.08em$ $ *máquina* $\color{SchemeDark}\kern0.08em\rangle$ $\color{SchemeDark}\langle\kern0.08em$ *etiqueta* $\color{SchemeDark}\kern0.08em\rangle$ $\color{SchemeDark}\langle\kern0.08em$ *n* $\color{SchemeDark}\kern0.08em\rangle$)

o eliminar todos los puntos de interrupción mediante

(cancel-all-breakpoints $\color{SchemeDark}\langle\kern0.08em$ $ *máquina* $\color{SchemeDark}\kern0.08em\rangle$)

Asignación de Almacenamiento y Recolección de Basura

En Section 5.4, mostraremos cómo implementar un evaluador de Scheme como una máquina de registros. Para simplificar la discusión, asumiremos que nuestras máquinas de registros pueden estar equipadas con una *memoria estructurada en listas*, en la cual las operaciones básicas para manipular datos estructurados en listas son primitivas. Postular la existencia de tal memoria es una abstracción útil cuando uno se enfoca en los mecanismos de control en un intérprete de Scheme, pero esto no refleja una visión realista de las operaciones de datos primitivas reales de las computadoras contemporáneas. Para obtener una imagen más completa de cómo opera un sistema Lisp, debemos investigar cómo se puede representar la estructura de lista de una manera que sea compatible con las memorias de computadora convencionales.

Hay dos consideraciones en la implementación de la estructura de lista. La primera es puramente un problema de representación: cómo representar la estructura de "caja y puntero" de los pares de Lisp, usando solo las capacidades de almacenamiento y direccionamiento de las memorias de computadora típicas. El segundo problema se refiere a la gestión de la memoria a medida que avanza un cálculo. El funcionamiento de un sistema Lisp depende crucialmente de la capacidad de crear continuamente nuevos objetos de datos. Estos incluyen objetos que son creados explícitamente por los procedimientos de Lisp que se interpretan, así como estructuras creadas por el propio intérprete, como entornos y listas de argumentos. Aunque la creación constante de nuevos objetos de datos no plantearía ningún problema en una computadora con una cantidad infinita de memoria rápidamente direccionable, las memorias de computadora solo están disponibles en tamaños finitos (qué lástima). Los sistemas Lisp, por lo tanto, proporcionan una facilidad de *asignación automática de almacenamiento* para soportar la ilusión de una memoria infinita. Cuando un objeto de datos ya no es necesario, la memoria asignada a él se recicla automáticamente y se utiliza para construir nuevos objetos de datos. Existen varias técnicas para proporcionar dicha asignación automática de almacenamiento. El método que discutiremos en esta sección se llama *recolección de basura*.

La Memoria como Vectores

Una memoria de computadora convencional puede considerarse como un arreglo de compartimentos, cada uno de los cuales puede contener una pieza de información. Cada compartimento tiene un nombre único, llamado su *dirección* o *ubicación*. Los sistemas de memoria típicos proporcionan dos operaciones primitivas: una que recupera los datos almacenados en una ubicación especificada y otra que asigna nuevos datos a una ubicación especificada. Las direcciones de memoria se pueden incrementar para admitir

el acceso secuencial a un conjunto de los compartimentos. Más generalmente, muchas operaciones de datos importantes requieren que las direcciones de memoria se traten como datos, que pueden almacenarse en ubicaciones de memoria y manipularse en los registros de la máquina. La representación de la estructura de lista es una aplicación de tal *aritmética de direcciones*.

Para modelar la memoria de la computadora, usamos un nuevo tipo de estructura de datos llamada *vector*. Abstractamente, un vector es un objeto de datos compuesto cuyos elementos individuales pueden ser accedidos por medio de un índice entero en una cantidad de tiempo que es independiente del índice.[5] Para describir las operaciones de memoria, usamos dos procedimientos primitivos de Scheme para manipular vectores:

- `(vector/ref` ⟨*vector*⟩ ⟨*n*⟩`)` devuelve el n^{th} elemento del vector.
- `(vector/set!` ⟨*vector*⟩ ⟨*n*⟩ ⟨*valor*⟩`)` establece el n^{th} elemento del vector al valor designado.

Por ejemplo, si `v` es un vector, entonces `(vector/ref v 5)` obtiene la quinta entrada en el vector `v` y `(vector/set! v 5 7)` cambia el valor de la quinta entrada del vector `v` a 7.[6] Para la memoria de la computadora, este acceso puede implementarse mediante el uso de aritmética de direcciones para combinar una *dirección base* que especifica la ubicación de inicio de un vector en la memoria con un *índice* que especifica el desplazamiento de un elemento particular del vector.

Representando datos de Lisp

Podemos usar vectores para implementar las estructuras de pares básicas requeridas para una memoria estructurada en listas. Imaginemos que la memoria de la computadora se divide en dos vectores: `the/cars` y `the/cdrs`. Representaremos la estructura de lista de la siguiente manera: Un puntero a un par es un índice en los dos vectores. El `car` del par es la entrada en `the/cars` con el índice designado, y el `cdr` del par es la entrada en `the/cdrs` con el índice designado. También necesitamos una representación para objetos que no son pares (como números y símbolos) y una forma de distinguir un tipo de dato de otro. Hay muchos métodos para lograr esto, pero todos se reducen a usar *punteros tipados*, es decir, a extender la noción de "puntero" para incluir información sobre el tipo de dato.[7] El tipo de dato permite al sistema distinguir un puntero a un par (que consiste en el tipo de dato "par" y un índice en los vectores de memoria) de punteros a otros tipos de datos (que consisten en algún otro tipo de dato y lo que sea que se esté usando para representar datos de ese tipo). Dos objetos de datos se consideran iguales (`eq?`) si sus punteros son idénticos.[8] La Figura 5.14 ilustra el uso de este

método para representar la lista `((1 2) 3 4)`, cuyo diagrama de caja y puntero también se muestra. Usamos prefijos de letras para denotar la información de tipo de dato. Por lo tanto, un puntero al par con índice 5 se denota `p5`, la lista vacía se denota por el puntero `e0`, y un puntero al número 4 se denota `n4`. En el diagrama de caja y puntero, hemos indicado en la parte inferior izquierda de cada par el índice del vector que especifica dónde se almacenan el `car` y el `cdr` del par. Las ubicaciones en blanco en `the/cars` y `the/cdrs` pueden contener partes de otras estructuras de lista (que no son de interés aquí).

Un puntero a un número, como `n4`, podría consistir en un tipo que indica datos numéricos junto con la representación real del número 4.[9] Para tratar con números que son demasiado grandes para ser representados en la cantidad fija de espacio asignado para un solo puntero, podríamos usar un tipo de dato *bignum* distinto, para el cual el puntero designa una lista en la que se almacenan las partes del número.[10]

Figura 5.14: Representaciones de diagrama de caja y puntero y de vector de memoria de la lista `((1 2) 3 4)`.

Un símbolo podría representarse como un puntero tipado que designa una secuencia de los caracteres que forman la representación impresa del símbolo. Esta secuencia es construida por el lector de Lisp cuando la cadena de caracteres se encuentra inicialmente en la entrada. Como queremos que dos instancias de un símbolo sean reconocidas como el "mismo" símbolo por `eq?` y queremos que `eq?` sea una prueba simple de igualdad de punteros, debemos asegurarnos de que si el lector ve la misma cadena de caracteres dos veces, utilizará el mismo puntero (a la misma secuencia de caracteres) para representar ambas ocurrencias. Para lograr esto, el lector mantiene una tabla, tradicionalmente llamada el *obarray*, de todos los símbolos que ha encontrado. Cuando el lector encuentra una cadena de caracteres y está a punto de construir un símbolo, verifica el obarray para ver si ha visto alguna vez la misma cadena de caracteres. Si no la ha visto, usa los caracteres para construir un nuevo símbolo (un puntero tipado a una nueva secuencia de caracteres) y e ingresa este puntero en el obarray. Si el lector ha visto la cadena antes, devuelve el puntero del símbolo almacenado en el obarray. Este proceso de reemplazar cadenas de caracteres por punteros únicos se llama *interning* de símbolos.

Implementando las operaciones primitivas de lista

Dado el esquema de representación anterior, podemos reemplazar cada

operación "primitiva" de lista de una máquina de registros con una o más operaciones primitivas de vector. Usaremos dos registros, `the/cars` y `the/cdrs`, para identificar los vectores de memoria, y asumiremos que `vector/ref` y `vector/set!` están disponibles como operaciones primitivas. También asumimos que las operaciones numéricas sobre punteros (como incrementar un puntero, usar un puntero de par para indexar un vector, o sumar dos números) usan solo la porción de índice del puntero tipado.

Por ejemplo, podemos hacer que una máquina de registros soporte las instrucciones

(assign < *reg* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle $ (op car) (reg < *reg* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$)) (assign < *reg* $\color{SchemeDark}_{\hbox{\ttfamily \scriptsize 1}}\rangle$ (op cdr) (reg < *reg* $\color{SchemeDark} _{\hbox{\ttfamily\scriptsize 2}}\rangle$))

si las implementamos, respectivamente, como

(assign < *reg* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle $ (op vector-ref) (reg the-cars) (reg < *reg* $\color{SchemeDark} _{\hbox{\ttfamily\scriptsize 2}}\rangle$)) (assign < *reg* $ \color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$ (op vector-ref) (reg the-cdrs) (reg < *reg* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$))

Las instrucciones

(perform (op set-car!) (reg < *reg* $\color{SchemeDark}_{\hbox{\ttfamily \scriptsize 1}}\rangle$) (reg < *reg* $\color{SchemeDark}_{\hbox{\ttfamily \scriptsize 2}}\rangle$)) (perform (op set-cdr!) (reg < *reg* $ \color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$) (reg < *reg* $ \color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$))

se implementan como

(perform (op vector-set!) (reg the-cars) (reg < *reg* $\color{SchemeDark} _{\hbox{\ttfamily\scriptsize 1}}\rangle$) (reg < *reg* $\color{SchemeDark} _{\hbox{\ttfamily\scriptsize 2}}\rangle$)) (perform (op vector-set!) (reg the-cdrs) (reg < *reg* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$) (reg < *reg* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$))

`cons` se realiza asignando un índice no utilizado y almacenando los argumentos de `cons` en `the/cars` y `the/cdrs` en esa posición de vector

indexada. Suponemos que hay un registro especial, `free`, que siempre contiene un puntero de par que contiene el siguiente índice disponible, y que podemos incrementar la parte del índice de ese puntero para encontrar la siguiente ubicación libre. [11] Por ejemplo, la instrucción

(assign < *reg* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle $ (op cons) (reg < *reg* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$) (reg < *reg* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 3}}\rangle$))

se implementa como la siguiente secuencia de operaciones de vector:[12]

(perform (op vector-set!) (reg the-cars) (reg free) (reg < *reg* $ \color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$)) (perform (op vector-set!) (reg the-cdrs) (reg free) (reg < *reg* $\color{SchemeDark} _{\hbox{\ttfamily\scriptsize 3}}\rangle$)) (assign < *reg* $ \color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$ (reg free)) (assign free (op +) (reg free) (const 1))

La operación `eq?`

(op eq?) (reg < *reg* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$) (reg < *reg* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$)

simplemente prueba la igualdad de todos los campos en los registros, y los predicados como `pair?`, `null?`, `symbol?` y `number?` solo necesitan verificar el campo de tipo.

Implementando pilas

Aunque nuestras máquinas de registros usan pilas, no necesitamos hacer nada especial aquí, ya que las pilas se pueden modelar en términos de listas. La pila puede ser una lista de los valores guardados, apuntada por un registro especial `the/stack`. Así, `(save` ⟨*reg*⟩`)` puede implementarse como

(assign the-stack (op cons) (reg < *reg* >) (reg the-stack))

Del mismo modo, `(restore` ⟨*reg*⟩`)` puede implementarse como

(assign < *reg* > (op car) (reg the-stack)) (assign the-stack (op cdr) (reg the-stack))

y `(perform (op initialize/stack))` puede implementarse como

(assign the-stack (const ()))

Estas operaciones se pueden expandir aún más en términos de las operaciones de vector dadas anteriormente. En las arquitecturas de computadora convencionales, sin embargo, suele ser ventajoso asignar la pila como un vector separado. Entonces, empujar y sacar de la pila (pushing and popping) puede lograrse incrementando o decrementando un índice en ese vector.

Ejercicio 5.20: Dibuje la representación de caja y puntero y la representación de vector de memoria (como en la Figura 5.14) de la estructura de lista producida por

(define x (cons 1 2)) (define y (list x x))

con el puntero `free` inicialmente en `p1`. ¿Cuál es el valor final de `free`? ¿Qué punteros representan los valores de `x` e `y`?

Ejercicio 5.21: Implemente máquinas de registros para los siguientes procedimientos. Asuma que las operaciones de memoria de estructura de lista están disponibles como primitivas de la máquina.

1. `count/leaves` recursivo:

   (define (count-leaves tree) (cond ((null? tree) 0) ((not (pair? tree)) 1) (else (+ (count-leaves (car tree)) (count-leaves (cdr tree))))))

2. `count/leaves` recursivo con contador explícito:

   (define (count-leaves tree) (define (count-iter tree n) (cond ((null? tree) n) ((not (pair? tree)) (+ n 1)) (else (count-iter (cdr tree) (count-iter (car tree) n))))) (count-iter tree 0))

Ejercicio 5.22: El Ejercicio 3.12 de Section 3.3.1 presentó un procedimiento `append` que anexa dos listas para formar una nueva lista y un procedimiento `append!` que empalma dos listas. Diseñe una máquina de registros para implementar cada uno de estos procedimientos. Asuma que las operaciones de memoria de estructura de lista están disponibles como operaciones primitivas.

Manteniendo la Ilusión de Memoria Infinita

El método de representación descrito en Section 5.3.1 resuelve el problema de implementar la estructura de lista, siempre que tengamos una cantidad infinita de memoria. Con una computadora real, eventualmente nos

quedaremos sin espacio libre en el que construir nuevos pares.[13] Sin embargo, la mayoría de los pares generados en un cálculo típico se usan solo para mantener resultados intermedios. Después de acceder a estos resultados, los pares ya no son necesarios—son *basura*. Por ejemplo, el cálculo

```
(accumulate + 0 (filter odd? (enumerate-interval 0 n)))
```

construye dos listas: la enumeración y el resultado de filtrar la enumeración. Cuando la acumulación se completa, estas listas ya no son necesarias, y la memoria asignada puede ser reclamada. Si podemos arreglárnoslas para recolectar toda la basura periódicamente, y si esto resulta en reciclar la memoria aproximadamente a la misma velocidad a la que construimos nuevos pares, habremos preservado la ilusión de que hay una cantidad infinita de memoria.

Para reciclar pares, debemos tener una forma de determinar qué pares asignados no son necesarios (en el sentido de que su contenido ya no puede influir en el futuro del cálculo). El método que examinaremos para lograr esto se conoce como *recolección de basura*. La recolección de basura se basa en la observación de que, en cualquier momento de una interpretación Lisp, los únicos objetos que pueden afectar el futuro del cálculo son aquellos a los que se puede llegar mediante alguna sucesión de operaciones `car` y `cdr` a partir de los punteros que se encuentran actualmente en los registros de la máquina.[14] Cualquier celda de memoria que no sea tan accesible puede ser reciclada.

Hay muchas formas de realizar la recolección de basura. El método que examinaremos aquí se llama *stop-and-copy*. La idea básica es dividir la memoria en dos mitades: "memoria de trabajo" y "memoria libre". Cuando `cons` construye pares, los asigna en la memoria de trabajo. Cuando la memoria de trabajo está llena, realizamos la recolección de basura localizando todos los pares útiles en la memoria de trabajo y copiándolos a ubicaciones consecutivas en la memoria libre. (Los pares útiles se localizan rastreando todos los punteros `car` y `cdr`, comenzando con los registros de la máquina). Como no copiamos la basura, presumiblemente habrá memoria libre adicional que podremos usar para asignar nuevos pares. Además, no se necesita nada en la memoria de trabajo, ya que todos los pares útiles en ella han sido copiados. Por lo tanto, si intercambiamos los roles de la memoria de trabajo y la memoria libre, podemos continuar procesando; se asignarán nuevos pares en la nueva memoria de trabajo (que era la antigua memoria libre). Cuando esta esté llena, podemos copiar los pares útiles en la nueva memoria libre (que era la antigua memoria de trabajo).[15]

Implementación de un recolector de basura stop-and-copy

Ahora usamos nuestro lenguaje de máquina de registros para describir el algoritmo stop-and-copy en más detalle. Asumiremos que hay un registro llamado `root` que contiene un puntero a una estructura que eventualmente apunta a todos los datos accesibles. Esto se puede arreglar almacenando el contenido de todos los registros de la máquina en una lista pre-asignada apuntada por `root` justo antes de comenzar la recolección de basura.[16] También asumimos que, además de la memoria de trabajo actual, hay memoria libre disponible en la que podemos copiar los datos útiles. La memoria de trabajo actual consiste en vectores cuyas direcciones base están en registros llamados `the/cars` y `the/cdrs`, y la memoria libre está en registros llamados `new/cars` y `new/cdrs`.

La recolección de basura se activa cuando agotamos las celdas libres en la memoria de trabajo actual, es decir, cuando una operación `cons` intenta incrementar el puntero `free` más allá del final del vector de memoria. Cuando el proceso de recolección de basura está completo, el puntero `root` apuntará a la nueva memoria, todos los objetos accesibles desde `root` habrán sido movidos a la nueva memoria, y el puntero `free` indicará el siguiente lugar en la nueva memoria donde se puede asignar un nuevo par. Además, los roles de la memoria de trabajo y la nueva memoria se habrán intercambiado—se construirán nuevos pares en la nueva memoria, comenzando en el lugar indicado por `free`, y la (anterior) memoria de trabajo estará disponible como la nueva memoria para la próxima recolección de basura. La Figura 5.15 muestra la disposición de la memoria justo antes y justo después de la recolección de basura.

Figura 5.15: Reconfiguración de la memoria por el proceso de recolección de basura.

El estado del proceso de recolección de basura se controla manteniendo dos punteros: `free` y `scan`. Estos se inicializan para apuntar al comienzo de la nueva memoria. El algoritmo comienza reubicando el par apuntado por `root` al comienzo de la nueva memoria. El par se copia, el puntero `root` se ajusta para apuntar a la nueva ubicación, y el puntero `free` se incrementa. Además, la antigua ubicación del par se marca para mostrar que su contenido ha sido movido. Esta marca se hace de la siguiente manera: En la posición `car`, colocamos una etiqueta especial que señala que este es un objeto ya movido. (Tal objeto se llama tradicionalmente un *corazón roto*.)[17] En la posición `cdr` colocamos una *dirección de reenvío* que apunta a la ubicación a la que el objeto ha sido movido.

Después de reubicar la raíz, el recolector de basura entra en su ciclo básico.

En cada paso del algoritmo, el puntero `scan` (inicialmente apuntando a la raíz reubicada) apunta a un par que ha sido movido a la nueva memoria pero cuyos punteros `car` y `cdr` todavía se refieren a objetos en la antigua memoria. Cada uno de estos objetos se reubica, y el puntero `scan` se incrementa. Para reubicar un objeto (por ejemplo, el objeto indicado por el puntero `car` del par que estamos escaneando) verificamos si el objeto ya ha sido movido (como lo indica la presencia de una etiqueta de corazón roto en la posición `car` del objeto). Si el objeto no ha sido movido, lo copiamos al lugar indicado por `free`, actualizamos `free`, establecemos un corazón roto en la antigua ubicación del objeto y actualizamos el puntero al objeto (en este ejemplo, el puntero `car` del par que estamos escaneando) para que apunte a la nueva ubicación. Si el objeto ya ha sido movido, su dirección de reenvío (que se encuentra en la posición `cdr` del corazón roto) se sustituye por el puntero en el par que se está escaneando. Eventualmente, todos los objetos accesibles habrán sido movidos y escaneados, momento en el cual el puntero `scan` alcanzará al puntero `free` y el proceso terminará.

Podemos especificar el algoritmo stop-and-copy como una secuencia de instrucciones para una máquina de registros. El paso básico de reubicar un objeto se logra mediante una subrutina llamada `relocate/old/result/in/new`. Esta subrutina obtiene su argumento, un puntero al objeto a reubicar, de un registro llamado `old`. Reubica el objeto designado (incrementando `free` en el proceso), pone un puntero al objeto reubicado en un registro llamado `new`, y regresa saltando al punto de entrada almacenado en el registro `relocate/continue`. Para comenzar la recolección de basura, invocamos esta subrutina para reubicar el puntero `root`, después de inicializar `free` y `scan`. Cuando se ha logrado la reubicación de `root`, instalamos el nuevo puntero como el nuevo `root` y entramos en el bucle principal del recolector de basura.

begin-garbage-collection (assign free (const 0)) (assign scan (const 0)) (assign old (reg root)) (assign relocate-continue (label reassign-root)) (goto (label relocate-old-result-in-new)) reassign-root (assign root (reg new)) (goto (label gc-loop))

En el bucle principal del recolector de basura debemos determinar si hay más objetos por escanear. Hacemos esto probando si el puntero `scan` coincide con el puntero `free`. Si los punteros son iguales, entonces todos los objetos accesibles han sido reubicados, y saltamos a `gc/flip`, que limpia las cosas para que podamos continuar el cálculo interrumpido. Si todavía hay pares por escanear, llamamos a la subrutina de reubicación para reubicar el `car` del siguiente par (colocando el puntero `car` en `old`). El registro `relocate/continue` se configura para que la subrutina regrese para actualizar el puntero `car`.

gc-loop (test (op =) (reg scan) (reg free)) (branch (label gc-flip)) (assign old (op vector-ref) (reg new-cars) (reg scan)) (assign relocate-continue (label update-car)) (goto (label relocate-old-result-in-new))

En `update/car`, modificamos el puntero `car` del par que se está escaneando, luego procedemos a reubicar el `cdr` del par. Regresamos a `update/cdr` cuando se ha logrado esa reubicación. Después de reubicar y actualizar el `cdr`, terminamos de escanear ese par, por lo que continuamos con el bucle principal.

update-car (perform (op vector-set!) (reg new-cars) (reg scan) (reg new)) (assign old (op vector-ref) (reg new-cdrs) (reg scan)) (assign relocate-continue (label update-cdr)) (goto (label relocate-old-result-in-new)) update-cdr (perform (op vector-set!) (reg new-cdrs) (reg scan) (reg new)) (assign scan (op +) (reg scan) (const 1)) (goto (label gc-loop))

La subrutina `relocate/old/result/in/new` reubica objetos de la siguiente manera: Si el objeto a reubicar (apuntado por `old`) no es un par, entonces devolvemos el mismo puntero al objeto sin cambios (en `new`). (Por ejemplo, podemos estar escaneando un par cuyo `car` es el número 4. Si representamos el `car` por `n4`, como se describe en Section 5.3.1, entonces queremos que el puntero `car` "reubicado" siga siendo `n4`.) De lo contrario, debemos realizar la reubicación. Si la posición `car` del par a reubicar contiene una etiqueta de corazón roto, entonces el par de hecho ya ha sido movido, por lo que recuperamos la dirección de reenvío (de la posición `cdr` del corazón roto) y devolvemos esto en `new`. Si el puntero en `old` apunta a un par aún no movido, entonces movemos el par a la primera celda libre en la nueva memoria (apuntada por `free`) y configuramos el corazón roto almacenando una etiqueta de corazón roto y una dirección de reenvío en la ubicación antigua. `relocate/old/result/in/new` utiliza un registro `oldcr` para mantener el `car` o el `cdr` del objeto apuntado por `old`.[18]

relocate-old-result-in-new (test (op pointer-to-pair?) (reg old)) (branch (label pair)) (assign new (reg old)) (goto (reg relocate-continue)) pair (assign oldcr (op vector-ref) (reg the-cars) (reg old)) (test (op broken-heart?) (reg oldcr)) (branch (label already-moved)) (assign new (reg free)) ; nueva ubicación para el par ;; Actualizar puntero `free`. (assign free (op +) (reg free) (const 1)) ;; Copiar el `car` y el `cdr` a la nueva memoria. (perform (op vector-set!) (reg new-cars) (reg new) (reg oldcr)) (assign oldcr (op vector-ref) (reg the-cdrs) (reg old)) (perform (op vector-set!) (reg new-cdrs) (reg new) (reg oldcr)) ;; Construir el corazón roto. (perform (op vector-set!) (reg the-cars) (reg old) (const broken-heart)) (perform (op vector-set!) (reg the-cdrs) (reg old) (reg new)) (goto (reg relocate-continue)) already-moved (assign new (op vector-ref) (reg the-cdrs) (reg old)) (goto (reg relocate-continue))

Al final del proceso de recolección de basura, intercambiamos el rol de las memorias antigua y nueva intercambiando punteros: intercambiando `the/cars` con `new/cars`, y `the/cdrs` con `new/cdrs`. Entonces estaremos listos para realizar otra recolección de basura la próxima vez que se agote la memoria.

gc-flip (assign temp (reg the-cdrs)) (assign the-cdrs (reg new-cdrs)) (assign new-cdrs (reg temp)) (assign temp (reg the-cars)) (assign the-cars (reg new-cars)) (assign new-cars (reg temp))

El Evaluador de Control Explícito

En Section 5.1 vimos cómo transformar programas simples de Scheme en descripciones de máquinas de registros. Ahora realizaremos esta transformación en un programa más complejo, el evaluador metacircular de Section 4.1.1–Section 4.1.4, que muestra cómo el comportamiento de un intérprete de Scheme puede describirse en términos de los procedimientos `eval` y `apply`. El *evaluador de control explícito* que desarrollamos en esta sección muestra cómo los mecanismos subyacentes de llamada a procedimientos y paso de argumentos utilizados en el proceso de evaluación pueden describirse en términos de operaciones sobre registros y pilas. Además, el evaluador de control explícito puede servir como una implementación de un intérprete de Scheme, escrito en un lenguaje que es muy similar al lenguaje de máquina nativo de las computadoras convencionales. El evaluador puede ser ejecutado por el simulador de máquina de registros de Section 5.2. Alternativamente, puede usarse como punto de partida para construir una implementación en lenguaje de máquina de un evaluador de Scheme, o incluso una máquina de propósito especial para evaluar expresiones de Scheme. La Figura 5.16 muestra tal implementación de hardware: un chip de silicio que actúa como un evaluador para Scheme. Los diseñadores del chip comenzaron con las especificaciones de la ruta de datos y del controlador para una máquina de registros similar al evaluador descrito en esta sección y usaron programas de automatización de diseño para construir el diseño del circuito integrado.[19]

Figura 5.16: Una implementación en chip de silicio de un evaluador para Scheme.

Registros y operaciones

Al diseñar el evaluador de control explícito, debemos especificar las operaciones que se utilizarán en nuestra máquina de registros. Describimos el evaluador metacircular en términos de sintaxis abstracta, utilizando procedimientos como `quoted?` y `make/procedure`. Al implementar la máquina de registros, podríamos expandir estos procedimientos en secuencias de operaciones elementales de memoria de estructura de lista, e implementar estas operaciones en nuestra máquina de registros. Sin embargo, esto haría que nuestro evaluador fuera muy largo, oscureciendo la estructura básica con detalles. Para aclarar la presentación, incluiremos como operaciones primitivas de la máquina de registros los procedimientos de sintaxis dados en Section 4.1.2 y los procedimientos para representar entornos y otros datos en tiempo de ejecución dados en las secciones Section 4.1.3 y Section 4.1.4. Para especificar completamente un evaluador que pudiera programarse en un lenguaje de máquina de bajo nivel o implementarse en hardware, reemplazaríamos estas operaciones por operaciones más elementales, utilizando la implementación de estructura de lista que describimos en Section 5.3.

Nuestra máquina de registros del evaluador de Scheme incluye una pila y siete registros: `exp`, `env`, `val`, `continue`, `proc`, `argl`, y `unev`. `exp` se usa para contener la expresión a ser evaluada, y `env` contiene el entorno en el que se debe realizar la evaluación. Al final de una evaluación, `val` contiene el valor obtenido al evaluar la expresión en el entorno designado. El registro `continue` se usa para implementar la recursión, como se explica en Section 5.1.4. (El evaluador necesita llamarse a sí mismo recursivamente, ya que evaluar una expresión requiere evaluar sus subexpresiones). Los registros `proc`, `argl` y `unev` se usan para evaluar combinaciones.

No proporcionaremos un diagrama de ruta de datos para mostrar cómo se conectan los registros y operaciones del evaluador, ni daremos la lista completa de operaciones de la máquina. Estas están implícitas en el controlador del evaluador, que se presentará en detalle.

El Núcleo del Evaluador de Control Explícito

El elemento central en el evaluador es la secuencia de instrucciones que comienza en `eval/dispatch`. Esto corresponde al procedimiento `eval` del evaluador metacircular descrito en Section 4.1.1. Cuando el controlador comienza en `eval/dispatch`, evalúa la expresión especificada por `exp` en el entorno especificado por `env`. Cuando la evaluación está completa, el controlador irá al punto de entrada almacenado en `continue`, y el registro `val` contendrá el valor de la expresión. Al igual que con el `eval` metacircular, la estructura de `eval/dispatch` es un análisis de casos sobre el tipo sintáctico

de la expresión a evaluar.[20]

eval-dispatch (test (op self-evaluating?) (reg exp)) (branch (label ev-self-eval)) (test (op variable?) (reg exp)) (branch (label ev-variable)) (test (op quoted?) (reg exp)) (branch (label ev-quoted)) (test (op assignment?) (reg exp)) (branch (label ev-assignment)) (test (op definition?) (reg exp)) (branch (label ev-definition)) (test (op if?) (reg exp)) (branch (label ev-if)) (test (op lambda?) (reg exp)) (branch (label ev-lambda)) (test (op begin?) (reg exp)) (branch (label ev-begin)) (test (op application?) (reg exp)) (branch (label ev-application)) (goto (label unknown-expression-type))

Evaluando expresiones simples

Los números y las cadenas (que son autoevaluables), las variables, las citas y las expresiones `lambda` no tienen subexpresiones que evaluar. Para estas, el evaluador simplemente coloca el valor correcto en el registro `val` y continúa la ejecución en el punto de entrada especificado por `continue`. La evaluación de expresiones simples se realiza con el siguiente código de controlador:

ev-self-eval (assign val (reg exp)) (goto (reg continue)) ev-variable (assign val (op lookup-variable-value) (reg exp) (reg env)) (goto (reg continue)) ev-quoted (assign val (op text-of-quotation) (reg exp)) (goto (reg continue)) ev-lambda (assign unev (op lambda-parameters) (reg exp)) (assign exp (op lambda-body) (reg exp)) (assign val (op make-procedure) (reg unev) (reg exp) (reg env)) (goto (reg continue))

Observe cómo `ev/lambda` usa los registros `unev` y `exp` para mantener los parámetros y el cuerpo de la expresión lambda para que puedan pasarse a la operación `make/procedure`, junto con el entorno en `env`.

Evaluando aplicaciones de procedimientos

Una aplicación de procedimiento se especifica mediante una combinación que contiene un operador y operandos. El operador es una subexpresión cuyo valor es un procedimiento, y los operandos son subexpresiones cuyos valores son los argumentos a los que se debe aplicar el procedimiento. El `eval` metacircular maneja las aplicaciones llamándose a sí mismo recursivamente para evaluar cada elemento de la combinación, y luego pasando los resultados a `apply`, que realiza la aplicación real del procedimiento. El evaluador de control explícito hace lo mismo; estas llamadas recursivas se implementan mediante instrucciones `goto`, junto con el uso de la pila para guardar registros que se restaurarán después de que regrese la llamada recursiva. Antes de cada llamada, tendremos cuidado de identificar qué

registros deben guardarse (porque sus valores se necesitarán más tarde).[21]

Comenzamos la evaluación de una aplicación evaluando el operador para producir un procedimiento, que luego se aplicará a los operandos evaluados. Para evaluar el operador, lo movemos al registro `exp` y vamos a `eval/dispatch`. El entorno en el registro `env` ya es el correcto en el que evaluar el operador. Sin embargo, guardamos `env` porque lo necesitaremos más tarde para evaluar los operandos. También extraemos los operandos en `unev` y lo guardamos en la pila. Configuramos `continue` para que `eval/dispatch` se reanude en `ev/appl/did/operator` después de que se haya evaluado el operador. Primero, sin embargo, guardamos el valor antiguo de `continue`, que le dice al controlador dónde continuar después de la aplicación.

ev-application (save continue) (save env) (assign unev (op operands) (reg exp)) (save unev) (assign exp (op operator) (reg exp)) (assign continue (label ev-appl-did-operator)) (goto (label eval-dispatch))

Al regresar de la evaluación de la subexpresión del operador, procedemos a evaluar los operandos de la combinación y a acumular los argumentos resultantes en una lista, contenida en `argl`. Primero restauramos los operandos no evaluados y el entorno. Inicializamos `argl` a una lista vacía. Luego asignamos al registro `proc` el procedimiento que se produjo al evaluar el operador. Si no hay operandos, vamos directamente a `apply/dispatch`. De lo contrario, guardamos `proc` en la pila y comenzamos el bucle de evaluación de argumentos:[22]

ev-appl-did-operator (restore unev) ; los operandos (restore env) (assign argl (op empty-arglist)) (assign proc (reg val)) ; el operador (test (op no-operands?) (reg unev)) (branch (label apply-dispatch)) (save proc)

Cada ciclo del bucle de evaluación de argumentos evalúa un operando de la lista en `unev` y acumula el resultado en `argl`. Para evaluar un operando, lo colocamos en el registro `exp` y vamos a `eval/dispatch`, después de establecer `continue` para que la ejecución se reanude con la fase de acumulación de argumentos. Pero primero guardamos los argumentos acumulados hasta ahora (contenidos en `argl`), el entorno (contenido en `env`), y los operandos restantes a evaluar (contenidos en `unev`). Se hace un caso especial para la evaluación del último operando, que se maneja en `ev/appl/last/arg`.

ev-appl-operand-loop (save argl) (assign exp (op first-operand) (reg unev)) (test (op last-operand?) (reg unev)) (branch (label ev-appl-last-arg)) (save env) (save unev) (assign continue (label ev-appl-accumulate-arg)) (goto (label eval-dispatch))

Cuando un operando ha sido evaluado, el valor se acumula en la lista contenida en `argl`. El operando se elimina luego de la lista de operandos no evaluados en `unev`, y la evaluación de argumentos continúa.

ev-appl-accumulate-arg (restore unev) (restore env) (restore argl) (assign argl (op adjoin-arg) (reg val) (reg argl)) (assign unev (op rest-operands) (reg unev)) (goto (label ev-appl-operand-loop))

La evaluación del último argumento se maneja de manera diferente. No hay necesidad de guardar el entorno o la lista de operandos no evaluados antes de ir a `eval/dispatch`, ya que no serán necesarios después de que se evalúe el último operando. Por lo tanto, regresamos de la evaluación a un punto de entrada especial `ev/appl/accum/last/arg`, que restaura la lista de argumentos, acumula el nuevo argumento, restaura el procedimiento guardado y se va a realizar la aplicación.[23]

ev-appl-last-arg (assign continue (label ev-appl-accum-last-arg)) (goto (label eval-dispatch)) ev-appl-accum-last-arg (restore argl) (assign argl (op adjoin-arg) (reg val) (reg argl)) (restore proc) (goto (label apply-dispatch))

Los detalles del bucle de evaluación de argumentos determinan el orden en el que el intérprete evalúa los operandos de una combinación (por ejemplo, de izquierda a derecha o de derecha a izquierda—ver Ejercicio 3.8). Este orden no está determinado por el evaluador metacircular, que hereda su estructura de control del Scheme subyacente en el que está implementado.[24] Porque el selector `first/operand` (usado en `ev/appl/operand/loop` para extraer operandos sucesivos de `unev`) se implementa como `car` y el selector `rest/operands` se implementa como `cdr`, el evaluador de control explícito evaluará los operandos de una combinación en orden de izquierda a derecha.

Aplicación de procedimientos

El punto de entrada `apply/dispatch` corresponde al procedimiento `apply` del evaluador metacircular. Para cuando llegamos a `apply/dispatch`, el registro `proc` contiene el procedimiento a aplicar y `argl` contiene la lista de argumentos evaluados a los que debe aplicarse. El valor guardado de `continue` (originalmente pasado a `eval/dispatch` y guardado en `ev/application`), que dice a dónde regresar con el resultado de la aplicación del procedimiento, está en la pila. Cuando la aplicación está completa, el controlador se transfiere al punto de entrada especificado por el `continue` guardado, con el resultado de la aplicación en `val`. Al igual que con el `apply` metacircular, hay dos casos a considerar. O bien el procedimiento a aplicar es una primitiva o es un procedimiento compuesto.

apply-dispatch (test (op primitive-procedure?) (reg proc)) (branch (label primitive-apply)) (test (op compound-procedure?) (reg proc)) (branch (label compound-apply)) (goto (label unknown-procedure-type))

Asumimos que cada primitiva se implementa para obtener sus argumentos de `argl` y colocar su resultado en `val`. Para especificar cómo la máquina maneja las primitivas, tendríamos que proporcionar una secuencia de instrucciones del controlador para implementar cada primitiva y hacer que `primitive/apply` despache a las instrucciones para la primitiva identificada por el contenido de `proc`. Como estamos interesados en la estructura del proceso de evaluación en lugar de los detalles de las primitivas, simplemente usaremos una operación `apply/primitive/procedure` que aplica el procedimiento en `proc` a los argumentos en `argl`. Con el propósito de simular el evaluador con el simulador de Section 5.2, usamos el procedimiento `apply/primitive/procedure`, que llama al sistema Scheme subyacente para realizar la aplicación, tal como lo hicimos para el evaluador metacircular en Section 4.1.4. Después de calcular el valor de la aplicación primitiva, restauramos `continue` y vamos al punto de entrada designado.

primitive-apply (assign val (op apply-primitive-procedure) (reg proc) (reg argl)) (restore continue) (goto (reg continue))

Para aplicar un procedimiento compuesto, procedemos igual que con el evaluador metacircular. Construimos un marco que liga los parámetros del procedimiento a los argumentos, usamos este marco para extender el entorno llevado por el procedimiento, y evaluamos en este entorno extendido la secuencia de expresiones que forma el cuerpo del procedimiento. `ev/sequence`, descrito a continuación en Section 5.4.2, maneja la evaluación de la secuencia.

compound-apply (assign unev (op procedure-parameters) (reg proc)) (assign env (op procedure-environment) (reg proc)) (assign env (op extend-environment) (reg unev) (reg argl) (reg env)) (assign unev (op procedure-body) (reg proc)) (goto (label ev-sequence))

`compound/apply` es el único lugar en el intérprete donde al registro `env` se le asigna un nuevo valor. Al igual que en el evaluador metacircular, el nuevo entorno se construye a partir del entorno llevado por el procedimiento, junto con la lista de argumentos y la lista correspondiente de variables a ser ligadas.

Evaluación de Secuencias y Recursión de Cola

La porción del evaluador de control explícito en `ev/sequence` es análoga al

procedimiento `eval/sequence` del evaluador metacircular. Maneja secuencias de expresiones en cuerpos de procedimientos o en expresiones explícitas `begin`.

Las expresiones explícitas `begin` se evalúan colocando la secuencia de expresiones a evaluar en `unev`, guardando `continue` en la pila, y saltando a `ev/sequence`.

ev-begin (assign unev (op begin-actions) (reg exp)) (save continue) (goto (label ev-sequence))

Las secuencias implícitas en los cuerpos de los procedimientos se manejan saltando a `ev/sequence` desde `compound/apply`, momento en el que `continue` ya está en la pila, habiendo sido guardado en `ev/application`.

Las entradas en `ev/sequence` y `ev/sequence/continue` forman un bucle que evalúa sucesivamente cada expresión en una secuencia. La lista de expresiones no evaluadas se mantiene en `unev`. Antes de evaluar cada expresión, verificamos si hay expresiones adicionales para ser evaluadas en la secuencia. Si es así, guardamos el resto de las expresiones no evaluadas (mantenidas en `unev`) y el entorno en el que deben ser evaluadas (mantenido en `env`) y llamamos a `eval/dispatch` para evaluar la expresión. Los dos registros guardados se restauran al regresar de esta evaluación, en `ev/sequence/continue`.

La expresión final en la secuencia se maneja de manera diferente, en el punto de entrada `ev/sequence/last/exp`. Como no hay más expresiones que evaluar después de esta, no necesitamos guardar `unev` o `env` antes de ir a `eval/dispatch`. El valor de toda la secuencia es el valor de la última expresión, así que después de la evaluación de la última expresión no hay nada más que hacer excepto continuar en el punto de entrada actualmente mantenido en la pila (que fue guardado por `ev/application` o `ev/begin`). En lugar de configurar `continue` para hacer que `eval/dispatch` regrese aquí y luego restaurar `continue` de la pila y continuar en ese punto de entrada, restauramos `continue` de la pila antes de ir a `eval/dispatch`, para que `eval/dispatch` continúe en ese punto de entrada después de evaluar la expresión.

ev-sequence (assign exp (op first-exp) (reg unev)) (test (op last-exp?) (reg unev)) (branch (label ev-sequence-last-exp)) (save unev) (save env) (assign continue (label ev-sequence-continue)) (goto (label eval-dispatch)) ev-sequence-continue (restore env) (restore unev) (assign unev (op rest-exps) (reg unev)) (goto (label ev-sequence)) ev-sequence-last-exp (restore continue) (goto (label eval-dispatch))

Recursión de cola

En el Capítulo 1 dijimos que el proceso descrito por un procedimiento como

(define (sqrt-iter guess x) (if (good-enough? guess x) guess (sqrt-iter (improve guess x) x)))

es un proceso iterativo. Aunque el procedimiento es sintácticamente recursivo (definido en términos de sí mismo), no es lógicamente necesario para un evaluador guardar información al pasar de una llamada a `sqrt/iter` a la siguiente.[25] Un evaluador que puede ejecutar un procedimiento como `sqrt/iter` sin requerir un aumento de almacenamiento a medida que el procedimiento continúa llamándose a sí mismo se llama un evaluador *recursivo de cola*. La implementación metacircular del evaluador en el Capítulo 4 no especifica si el evaluador es recursivo de cola, porque ese evaluador hereda su mecanismo para guardar estado del Scheme subyacente. Con el evaluador de control explícito, sin embargo, podemos rastrear el proceso de evaluación para ver cuándo las llamadas a procedimientos causan una acumulación neta de información en la pila.

Nuestro evaluador es recursivo de cola, porque para evaluar la expresión final de una secuencia nos transferimos directamente a `eval/dispatch` sin guardar ninguna información en la pila. Por lo tanto, evaluar la expresión final en una secuencia—incluso si es una llamada a un procedimiento (como en `sqrt/iter`, donde la expresión `if`, que es la última expresión en el cuerpo del procedimiento, se reduce a una llamada a `sqrt/iter`)—no causará que se acumule ninguna información en la pila.[26]

Si no pensáramos en aprovechar el hecho de que era innecesario guardar información en este caso, podríamos haber implementado `eval/sequence` tratando todas las expresiones en una secuencia de la misma manera—guardando los registros, evaluando la expresión, regresando para restaurar los registros, y repitiendo esto hasta que todas las expresiones hayan sido evaluadas:[27]

ev-sequence (test (op no-more-exps?) (reg unev)) (branch (label ev-sequence-end)) (assign exp (op first-exp) (reg unev)) (save unev) (save env) (assign continue (label ev-sequence-continue)) (goto (label eval-dispatch)) ev-sequence-continue (restore env) (restore unev) (assign unev (op rest-exps) (reg unev)) (goto (label ev-sequence)) ev-sequence-end (restore continue) (goto (reg continue))

Esto puede parecer un cambio menor a nuestro código anterior para la evaluación de una secuencia: La única diferencia es que pasamos por el ciclo

de guardar-restaurar para la última expresión en una secuencia, así como para las demás. El intérprete seguirá dando el mismo valor para cualquier expresión. Pero este cambio es fatal para la implementación recursiva de cola, porque ahora debemos regresar después de evaluar la expresión final en una secuencia para deshacer los guardados de registro (inútiles). Estos guardados adicionales se acumularán durante un anidamiento de llamadas a procedimientos. Consecuentemente, procesos como `sqrt/iter` requerirán un espacio proporcional al número de iteraciones en lugar de requerir un espacio constante. Esta diferencia puede ser significativa. Por ejemplo, con la recursión de cola, un bucle infinito puede expresarse usando solo el mecanismo de llamada a procedimiento:

(define (count n) (newline) (display n) (count (+ n 1)))

Sin recursión de cola, tal procedimiento eventualmente se quedaría sin espacio en la pila, y expresar una verdadera iteración requeriría algún mecanismo de control distinto de la llamada a procedimiento.

Condicionales, Asignaciones y Definiciones

Al igual que con el evaluador metacircular, las formas especiales se manejan evaluando selectivamente fragmentos de la expresión. Para una expresión `if`, debemos evaluar el predicado y decidir, basándose en el valor del predicado, si evaluar el consecuente o la alternativa.

Antes de evaluar el predicado, guardamos la propia expresión `if` para que podamos extraer más tarde el consecuente o la alternativa. También guardamos el entorno, que necesitaremos más tarde para evaluar el consecuente o la alternativa, y guardamos `continue`, que necesitaremos más tarde para volver a la evaluación de la expresión que está esperando el valor del `if`.

ev-if (save exp) ; guardar expresión para más tarde (save env) (save continue) (assign continue (label ev-if-decide)) (assign exp (op if-predicate) (reg exp)) (goto (label eval-dispatch)) ; evaluar el predicado

Cuando regresamos de evaluar el predicado, probamos si fue verdadero o falso y, dependiendo del resultado, colocamos ya sea el consecuente o la alternativa en `exp` antes de ir a `eval/dispatch`. Note que restaurar `env` y `continue` aquí configura `eval/dispatch` para tener el entorno correcto y para continuar en el lugar correcto para recibir el valor de la expresión `if`.

ev-if-decide (restore continue) (restore env) (restore exp) (test (op true?) (reg val)) (branch (label ev-if-consequent)) ev-if-alternative (assign exp (op if-

alternative) (reg exp)) (goto (label eval-dispatch)) ev-if-consequent (assign exp (op if-consequent) (reg exp)) (goto (label eval-dispatch))

Asignaciones y definiciones

Las asignaciones son manejadas por `ev/assignment`, al que se llega desde `eval/dispatch` con la expresión de asignación en `exp`. El código en `ev/assignment` primero evalúa la parte del valor de la expresión y luego instala el nuevo valor en el entorno. Se asume que `set/variable/value!` está disponible como una operación de la máquina.

ev-assignment (assign unev (op assignment-variable) (reg exp)) (save unev) ; guardar variable para más tarde (assign exp (op assignment-value) (reg exp)) (save env) (save continue) (assign continue (label ev-assignment-1)) (goto (label eval-dispatch)) ; evaluar el valor de la asignación ev-assignment-1 (restore continue) (restore env) (restore unev) (perform (op set-variable-value!) (reg unev) (reg val) (reg env)) (assign val (const ok)) (goto (reg continue))

Las definiciones se manejan de manera similar:

ev-definition (assign unev (op definition-variable) (reg exp)) (save unev) ; guardar variable para más tarde (assign exp (op definition-value) (reg exp)) (save env) (save continue) (assign continue (label ev-definition-1)) (goto (label eval-dispatch)) ; evaluar el valor de la definición ev-definition-1 (restore continue) (restore env) (restore unev) (perform (op define-variable!) (reg unev) (reg val) (reg env)) (assign val (const ok)) (goto (reg continue))

> **Ejercicio 5.23:** Extienda el evaluador para manejar expresiones derivadas como `cond`, `let`, etc. (Section 4.1.2). Puede "hacer trampa" y asumir que los transformadores de sintaxis como `cond/>if` están disponibles como operaciones de la máquina.[28]
>
> **Ejercicio 5.24:** Implemente `cond` como una nueva forma especial básica sin reducirla a `if`. Tendrá que construir un bucle que pruebe los predicados de las cláusulas `cond` sucesivas hasta que encuentre una que sea verdadera, y luego usar `ev/sequence` para evaluar las acciones de la cláusula.
>
> **Ejercicio 5.25:** Modifique el evaluador para que use evaluación de orden normal, basado en el evaluador perezoso de Section 4.2.

Ejecutando el Evaluador

Con la implementación del evaluador de control explícito llegamos al final de un desarrollo, comenzado en el Capítulo 1, en el que hemos explorado sucesivamente modelos más precisos del proceso de evaluación. Comenzamos con el modelo de sustitución relativamente informal, luego lo extendimos en el Capítulo 3 al modelo de entorno, que nos permitió tratar con estado y cambio. En el evaluador metacircular del Capítulo 4, usamos Scheme mismo como lenguaje para hacer más explícita la estructura del entorno construida durante la evaluación de una expresión. Ahora, con las máquinas de registros, hemos echado un vistazo de cerca a los mecanismos del evaluador para la gestión del almacenamiento, el paso de argumentos y el control. En cada nuevo nivel de descripción, hemos tenido que plantear cuestiones y resolver ambigüedades que no eran aparentes en el tratamiento anterior, menos preciso, de la evaluación. Para entender el comportamiento del evaluador de control explícito, podemos simularlo y monitorear su rendimiento.

Instalaremos un bucle de conductor en nuestra máquina evaluadora. Esto juega el papel del procedimiento `driver/loop` de Section 4.1.4. El evaluador imprimirá repetidamente un prompt, leerá una expresión, evaluará la expresión yendo a `eval/dispatch`, e imprimirá el resultado. Las siguientes instrucciones forman el comienzo de la secuencia del controlador del evaluador de control explícito:[29]

read-eval-print-loop (perform (op initialize-stack)) (perform (op prompt-for-input) (const ";;Entrada EC-Eval:")) (assign exp (op read)) (assign env (op get-global-environment)) (assign continue (label print-result)) (goto (label eval-dispatch)) print-result (perform (op announce-output) (const ";;Valor EC-Eval:")) (perform (op user-print) (reg val)) (goto (label read-eval-print-loop))

Cuando encontramos un error en un procedimiento (como el "error de tipo de procedimiento desconocido" indicado en `apply/dispatch`), imprimimos un mensaje de error y volvemos al bucle del conductor.[30]

unknown-expression-type (assign val (const unknown-expression-type-error)) (goto (label signal-error)) unknown-procedure-type (restore continue) ; limpiar pila (de `apply/dispatch`) (assign val (const unknown-procedure-type-error)) (goto (label signal-error)) signal-error (perform (op user-print) (reg val)) (goto (label read-eval-print-loop))

Para los propósitos de la simulación, inicializamos la pila cada vez que pasamos por el bucle del conductor, ya que podría no estar vacía después de un error (como una variable no definida) que interrumpe una evaluación.[31]

Si combinamos todos los fragmentos de código presentados en Section 5.4.1–

Section 5.4.4, podemos crear un modelo de máquina evaluadora que podemos ejecutar usando el simulador de máquina de registros de Section 5.2.

(define eceval (make-machine '(exp env val proc argl continue unev) eceval-operations '(read-eval-print-loop ⟨ *controlador completo de la máquina como se dio arriba* ⟩)))

Debemos definir procedimientos de Scheme para simular las operaciones utilizadas como primitivas por el evaluador. Estos son los mismos procedimientos que usamos para el evaluador metacircular en Section 4.1, junto con los pocos adicionales definidos en las notas a pie de página a lo largo de Section 5.4.

(define eceval-operations (list (list 'self-evaluating? self-evaluating) ⟨ *lista completa de operaciones para la máquina eceval* ⟩))

Finalmente, podemos inicializar el entorno global y ejecutar el evaluador:

(define the-global-environment (setup-environment)) (start eceval) *;;; Entrada EC-Eval:* (define (append x y) (if (null? x) y (cons (car x) (append (cdr x) y)))) *;;; Valor EC-Eval: ok ;;; Entrada EC-Eval:* (append '(a b c) '(d e f)) *;;; Valor EC-Eval: (a b c d e f)*

Por supuesto, evaluar expresiones de esta manera tomará mucho más tiempo que si las hubiéramos escrito directamente en Scheme, debido a los múltiples niveles de simulación involucrados. Nuestras expresiones son evaluadas por la máquina del evaluador de control explícito, que está siendo simulada por un programa de Scheme, que a su vez está siendo evaluado por el intérprete de Scheme.

Monitoreando el rendimiento del evaluador

La simulación puede ser una herramienta poderosa para guiar la implementación de evaluadores. Las simulaciones facilitan no solo explorar variaciones del diseño de la máquina de registros, sino también monitorear el rendimiento del evaluador simulado. Por ejemplo, un factor importante en el rendimiento es cuán eficientemente el evaluador usa la pila. Podemos observar el número de operaciones de pila requeridas para evaluar diversas expresiones definiendo la máquina de registros del evaluador con la versión del simulador que recopila estadísticas sobre el uso de la pila (Section 5.2.4), y agregando una instrucción en el punto de entrada `print/result` del evaluador para imprimir las estadísticas:

print-result (perform (op print-stack-statistics)) ; instrucción añadida
(perform (op announce-output) (const ";;; Valor EC-Eval:")) … ; igual que antes

Las interacciones con el evaluador ahora se ven así:

;;; Entrada EC-Eval: (define (factorial n) (if (= n 1) 1 (* (factorial (- n 1)) n))) *(total-pushes = 3 maximum-depth = 3) ;;; Valor EC-Eval: ok ;;; Entrada EC-Eval:* (factorial 5) *(total-pushes = 144 maximum-depth = 28) ;;; Valor EC-Eval: 120*

Tenga en cuenta que el bucle conductor del evaluador reinicializa la pila al comienzo de cada interacción, de modo que las estadísticas impresas se referirán solo a las operaciones de pila utilizadas para evaluar la expresión anterior.

Ejercicio 5.26: Use la pila monitoreada para explorar la propiedad de recursión de cola del evaluador (Section 5.4.2). Inicie el evaluador y defina el procedimiento iterativo `factorial` de Section 1.2.1:

(define (factorial n) (define (iter product counter) (if (> counter n) product (iter (* counter product) (+ counter 1)))) (iter 1 1))

Ejecute el procedimiento con algunos valores pequeños de *n*. Registre la profundidad máxima de la pila y el número de pushes requeridos para calcular *n*! para cada uno de estos valores.

1. Encontrará que la profundidad máxima requerida para evaluar *n*! es independiente de *n*. ¿Cuál es esa profundidad?

2. Determine a partir de sus datos una fórmula en términos de *n* para el número total de operaciones de push utilizadas en la evaluación de *n*! para cualquier $n \geq 1$. Tenga en cuenta que el número de operaciones utilizadas es una función lineal de *n* y, por lo tanto, está determinado por dos constantes.

Ejercicio 5.27: Para comparar con el Ejercicio 5.26, explore el comportamiento del siguiente procedimiento para calcular factoriales recursivamente:

(define (factorial n) (if (= n 1) 1 (* (factorial (- n 1)) n)))

Ejecutando este procedimiento con la pila monitoreada, determine, como función de *n*, la profundidad máxima de la pila y el número total de pushes utilizados en la evaluación de *n*! para $n \geq 1$. (Nuevamente, estas

funciones serán lineales.) Resuma sus experimentos llenando la siguiente tabla con las expresiones apropiadas en términos de *n*:

$$\vbox{ \offinterlineskip \halign{ \strut \hfil \quad #\quad \hfil & \vrule \hfil \quad #\quad \hfil & \vrule \hfil \quad #\quad \hfil \cr & Profundidad máxima & Número de pushes \cr \noalign{\hrule} Factorial & & \cr recursivo & & \cr \noalign{\hrule} Factorial & & \cr iterativo & & \cr } }$$

La profundidad máxima es una medida de la cantidad de espacio utilizado por el evaluador para llevar a cabo el cálculo, y el número de pushes se correlaciona bien con el tiempo requerido.

Ejercicio 5.28: Modifique la definición del evaluador cambiando `eval/sequence` como se describe en Section 5.4.2 para que el evaluador ya no sea recursivo de cola. Vuelva a ejecutar sus experimentos de Ejercicio 5.26 y Ejercicio 5.27 para demostrar que ambas versiones del procedimiento `factorial` ahora requieren un espacio que crece linealmente con su entrada.

Ejercicio 5.29: Monitoree las operaciones de la pila en el cálculo de Fibonacci recursivo de árbol:

(define (fib n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))

1. Dé una fórmula en términos de *n* para la profundidad máxima de la pila requerida para calcular ${\rm Fib}(n)$ para $n \geq 2$. Pista: En la Sección 1.2.2 argumentamos que el espacio utilizado por este proceso crece linealmente con *n*.

2. Dé una fórmula para el número total de pushes utilizados para calcular ${\rm Fib}(n)$ para $n \geq 2$. Debería encontrar que el número de pushes (que se correlaciona bien con el tiempo utilizado) crece exponencialmente con *n*. Pista: Sea $S(n)$ el número de pushes utilizados para calcular ${\rm Fib}(n)$. Debería ser capaz de argumentar que existe una fórmula que expresa $S(n)$ en términos de $S(n-1)$, $S(n-2)$, y alguna constante de "sobrecarga" fija k que es independiente de *n*. Dé la fórmula, y diga qué es k. Luego muestre que $S(n)$ puede expresarse como $a\cdot{\rm Fib}(n+1)+b$ y dé los valores de a y b.

Ejercicio 5.30: Nuestro evaluador actualmente captura y señala solo dos tipos de errores—tipos de expresión desconocidos y tipos de procedimiento desconocidos. Otros errores nos sacarán del bucle de lectura-evaluación-impresión del evaluador. Cuando ejecutamos el

evaluador usando el simulador de máquina de registros, estos errores son capturados por el sistema Scheme subyacente. Esto es análogo a que la computadora se bloquee cuando un programa de usuario comete un error.[32] Es un gran proyecto hacer que un sistema de errores real funcione, pero vale la pena el esfuerzo entender lo que está involucrado aquí.

1. Los errores que ocurren en el proceso de evaluación, como un intento de acceder a una variable no ligada, podrían ser capturados cambiando la operación de búsqueda para que devuelva un código de condición distinguido, que no puede ser un valor posible de ninguna variable de usuario. El evaluador puede probar este código de condición y luego hacer lo necesario para ir a `signal/error`. Encuentre todos los lugares en el evaluador donde tal cambio es necesario y arréglelos. Esto es mucho trabajo.

2. Mucho peor es el problema de manejar errores que son señalados al aplicar procedimientos primitivos, como un intento de dividir por cero o un intento de extraer el `car` de un símbolo. En un sistema de alta calidad escrito profesionalmente, cada aplicación primitiva se verifica por seguridad como parte de la primitiva. Por ejemplo, cada llamada a `car` podría primero verificar que el argumento sea un par. Si el argumento no es un par, la aplicación devolvería un código de condición distinguido al evaluador, que luego informaría el fallo. Podríamos arreglar esto en nuestro simulador de máquina de registros haciendo que cada procedimiento primitivo verifique la aplicabilidad y devuelva un código de condición distinguido apropiado en caso de fallo. Entonces, el código `primitive/apply` en el evaluador puede verificar el código de condición e ir a `signal/error` si es necesario. Construya esta estructura y hágala funcionar. Este es un proyecto importante.

Compilación

El evaluador de control explícito de Section 5.4 es una máquina de registros cuyo controlador interpreta programas de Scheme. En esta sección veremos cómo ejecutar programas de Scheme en una máquina de registros cuyo controlador no es un intérprete de Scheme.

La máquina del evaluador de control explícito es universal—puede llevar a cabo cualquier proceso computacional que pueda describirse en Scheme. El controlador del evaluador orquesta el uso de sus rutas de datos para realizar la computación deseada. Por lo tanto, las rutas de datos del evaluador son

universales: son suficientes para realizar cualquier computación que deseemos, dado un controlador apropiado.[33]

Las computadoras comerciales de propósito general son máquinas de registros organizadas en torno a una colección de registros y operaciones que constituyen un conjunto universal eficiente y conveniente de rutas de datos. El controlador para una máquina de propósito general es un intérprete para un lenguaje de máquina de registros como el que hemos estado usando. Este lenguaje se llama el *lenguaje nativo* de la máquina, o simplemente *lenguaje de máquina*. Los programas escritos en lenguaje de máquina son secuencias de instrucciones que usan las rutas de datos de la máquina. Por ejemplo, la secuencia de instrucciones del evaluador de control explícito puede considerarse como un programa en lenguaje de máquina para una computadora de propósito general en lugar de como el controlador para una máquina intérprete especializada.

Hay dos estrategias comunes para cerrar la brecha entre los lenguajes de alto nivel y los lenguajes de máquina de registros. El evaluador de control explícito ilustra la estrategia de interpretación. Un intérprete escrito en el lenguaje nativo de una máquina configura la máquina para ejecutar programas escritos en un lenguaje (llamado el *lenguaje fuente*) que puede diferir del lenguaje nativo de la máquina que realiza la evaluación. Los procedimientos primitivos del lenguaje fuente se implementan como una biblioteca de subrutinas escritas en el lenguaje nativo de la máquina dada. Un programa a ser interpretado (llamado el *programa fuente*) se representa como una estructura de datos. El intérprete recorre esta estructura de datos, analizando el programa fuente. A medida que lo hace, simula el comportamiento previsto del programa fuente llamando a subrutinas primitivas apropiadas de la biblioteca.

En esta sección, exploramos la estrategia alternativa de *compilación*. Un compilador para un lenguaje fuente y una máquina dados traduce un programa fuente en un programa equivalente (llamado el *programa objeto*) escrito en el lenguaje nativo de la máquina. El compilador que implementamos en esta sección traduce programas escritos en Scheme a secuencias de instrucciones para ser ejecutadas usando las rutas de datos de la máquina del evaluador de control explícito. [34]

En comparación con la interpretación, la compilación puede proporcionar un gran aumento en la eficiencia de la ejecución del programa, como explicaremos a continuación en la descripción general del compilador. Por otro lado, un intérprete proporciona un entorno más potente para el desarrollo interactivo de programas y la depuración, porque el programa fuente que se está ejecutando está disponible en tiempo de ejecución para ser

examinado y modificado. Además, debido a que toda la biblioteca de primitivas está presente, se pueden construir nuevos programas y agregarlos al sistema durante la depuración.

En vista de las ventajas complementarias de la compilación y la interpretación, los entornos modernos de desarrollo de programas persiguen una estrategia mixta. Los intérpretes de Lisp generalmente están organizados para que los procedimientos interpretados y los procedimientos compilados puedan llamarse entre sí. Esto permite a un programador compilar aquellas partes de un programa que se asumen depuradas, obteniendo así la ventaja de eficiencia de la compilación, mientras se retiene el modo interpretativo de ejecución para aquellas partes del programa que están en el flujo del desarrollo interactivo y la depuración. En Section 5.5.7, después de que hayamos implementado el compilador, mostraremos cómo interconectarlo con nuestro intérprete para producir un sistema de desarrollo intérprete-compilador integrado.

Un vistazo general al compilador

Nuestro compilador es muy parecido a nuestro intérprete, tanto en su estructura como en la función que realiza. En consecuencia, los mecanismos utilizados por el compilador para analizar expresiones serán similares a los utilizados por el intérprete. Además, para facilitar la interconexión de código compilado e interpretado, diseñaremos el compilador para generar código que obedezca las mismas convenciones de uso de registros que el intérprete: El entorno se mantendrá en el registro `env`, las listas de argumentos se acumularán en `argl`, un procedimiento a aplicar estará en `proc`, los procedimientos devolverán sus respuestas en `val`, y la ubicación a la que un procedimiento debe regresar se mantendrá en `continue`. En general, el compilador traduce un programa fuente en un programa objeto que realiza esencialmente las mismas operaciones de registro que realizaría el intérprete al evaluar el mismo programa fuente.

Esta descripción sugiere una estrategia para implementar un compilador rudimentario: Recorremos la expresión de la misma manera que lo hace el intérprete. Cuando encontramos una instrucción de registro que el intérprete realizaría al evaluar la expresión, no ejecutamos la instrucción, sino que la acumulamos en una secuencia. La secuencia resultante de instrucciones será el código objeto. Observe la ventaja de eficiencia de la compilación sobre la interpretación. Cada vez que el intérprete evalúa una expresión—por ejemplo, `(f 84 96)`—realiza el trabajo de clasificar la expresión (descubriendo que es una aplicación de procedimiento) y probar el final de la lista de operandos (descubriendo que hay dos operandos). Con un compilador, la expresión se analiza solo una vez, cuando se genera la

secuencia de instrucciones en tiempo de compilación. El código objeto producido por el compilador contiene solo las instrucciones que evalúan el operador y los dos operandos, ensamblan la lista de argumentos y aplican el procedimiento (en `proc`) a los argumentos (en `argl`).

Este es el mismo tipo de optimización que implementamos en el evaluador analizador de Section 4.1.7. Pero hay más oportunidades para ganar eficiencia en el código compilado. A medida que el intérprete se ejecuta, sigue un proceso que debe ser aplicable a cualquier expresión en el lenguaje. En contraste, un segmento dado de código compilado está destinado a ejecutar alguna expresión particular. Esto puede marcar una gran diferencia, por ejemplo, en el uso de la pila para guardar registros. Cuando el intérprete evalúa una expresión, debe estar preparado para cualquier contingencia. Antes de evaluar una subexpresión, el intérprete guarda todos los registros que se necesitarán más tarde, porque la subexpresión podría requerir una evaluación arbitraria. Un compilador, por otro lado, puede explotar la estructura de la expresión particular que está procesando para generar código que evite operaciones de pila innecesarias.

Como ejemplo, considere la combinación `(f 84 96)`. Antes de que el intérprete evalúe el operador de la combinación, se prepara para esta evaluación guardando los registros que contienen los operandos y el entorno, cuyos valores se necesitarán más tarde. El intérprete luego evalúa el operador para obtener el resultado en `val`, restaura los registros guardados y finalmente mueve el resultado de `val` a `proc`. Sin embargo, en la expresión particular que estamos tratando, el operador es el símbolo `f`, cuya evaluación se logra mediante la operación de la máquina `lookup/variable/value`, que no altera ningún registro. El compilador que implementamos en esta sección aprovechará este hecho y generará código que evalúa el operador usando la instrucción

(assign proc (op lookup-variable-value) (const f) (reg env))

Este código no solo evita los innecesarios guardados y restauraciones, sino que también asigna el valor de la búsqueda directamente a `proc`, mientras que el intérprete obtendría el resultado en `val` y luego lo movería a `proc`.

Un compilador también puede optimizar el acceso al entorno. Habiendo analizado el código, el compilador puede en muchos casos saber en qué marco se ubicará una variable particular y acceder a ese marco directamente, en lugar de realizar la búsqueda `lookup/variable/value`. Discutiremos cómo implementar dicho acceso a variables en Section 5.5.6. Hasta entonces, sin embargo, nos centraremos en el tipo de optimizaciones de registro y pila descritas anteriormente. Hay muchas otras optimizaciones que puede realizar

un compilador, como codificar operaciones primitivas "en línea" en lugar de usar un mecanismo general de `apply` (ver Ejercicio 5.38); pero no las enfatizaremos aquí. Nuestro principal objetivo en esta sección es ilustrar el proceso de compilación en un contexto simplificado (pero aún interesante).

Estructura del Compilador

En Section 4.1.7 modificamos nuestro intérprete metacircular original para separar el análisis de la ejecución. Analizamos cada expresión para producir un procedimiento de ejecución que tomaba un entorno como argumento y realizaba las operaciones requeridas. En nuestro compilador, haremos esencialmente el mismo análisis. Sin embargo, en lugar de producir procedimientos de ejecución, generaremos secuencias de instrucciones para ser ejecutadas por nuestra máquina de registros.

El procedimiento `compile` es el despacho de nivel superior en el compilador. Corresponde al procedimiento `eval` de Section 4.1.1, al procedimiento `analyze` de Section 4.1.7, y al punto de entrada `eval/dispatch` del evaluador de control explícito en Section 5.4.1. El compilador, al igual que los intérpretes, utiliza los procedimientos de sintaxis de expresiones definidos en Section 4.1.2.[35] `compile` realiza un análisis de casos sobre el tipo sintáctico de la expresión a compilar. Para cada tipo de expresión, despacha a un *generador de código* especializado:

(define (compile exp target linkage) (cond ((self-evaluating? exp) (compile-self-evaluating exp target linkage)) ((quoted? exp) (compile-quoted exp target linkage)) ((variable? exp) (compile-variable exp target linkage)) ((assignment? exp) (compile-assignment exp target linkage)) ((definition? exp) (compile-definition exp target linkage)) ((if? exp) (compile-if exp target linkage)) ((lambda? exp) (compile-lambda exp target linkage)) ((begin? exp) (compile-sequence (begin-actions exp) target linkage)) ((cond? exp) (compile (cond->if exp) target linkage)) ((application? exp) (compile-application exp target linkage)) (else (error "Unknown expression type: COMPILE" exp))))

Destinos y enlaces

`compile` y los generadores de código que llama toman dos argumentos además de la expresión a compilar. Hay un *destino*, que especifica el registro en el que el código compilado debe devolver el valor de la expresión. También hay un *descriptor de enlace*, que describe cómo el código resultante de la compilación de la expresión debe proceder cuando haya terminado su ejecución. El descriptor de enlace puede requerir que el código haga una de las siguientes tres cosas:

- continuar en la siguiente instrucción en secuencia (esto se especifica con el descriptor de enlace `next`),
- regresar del procedimiento que se está compilando (esto se especifica con el descriptor de enlace `return`), o
- saltar a un punto de entrada con nombre (esto se especifica usando la etiqueta designada como descriptor de enlace).

Por ejemplo, compilar la expresión `5` (que es autoevaluable) con un destino del registro `val` y un enlace de `next` debería producir la instrucción

(assign val (const 5))

Compilar la misma expresión con un enlace de `return` debería producir las instrucciones

(assign val (const 5)) (goto (reg continue))

En el primer caso, la ejecución continuará con la siguiente instrucción en la secuencia. En el segundo caso, regresaremos de una llamada a procedimiento. En ambos casos, el valor de la expresión se colocará en el registro de destino `val`.

Secuencias de instrucciones y uso de la pila

Cada generador de código devuelve una *secuencia de instrucciones* que contiene el código objeto que ha generado para la expresión. La generación de código para una expresión compuesta se logra combinando la salida de generadores de código más simples para las expresiones componentes, así como la evaluación de una expresión compuesta se logra evaluando las expresiones componentes.

El método más simple para combinar secuencias de instrucciones es un procedimiento llamado `append/instruction/sequences`. Toma como argumentos cualquier número de secuencias de instrucciones que deben ejecutarse secuencialmente; las anexa y devuelve la secuencia combinada. Es decir, si $\langle seq_1 \rangle$ y $\langle seq_2 \rangle$ son secuencias de instrucciones, entonces evaluar

(append-instruction-sequences < *seq* $\color{SchemeDark} _{\hbox{\ttfamily\scriptsize 1}}\rangle$ < *seq* $\color{SchemeDark} _{\hbox{\ttfamily\scriptsize 2}}\rangle$)

produce la secuencia

$\langle$ *seq* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$
$\langle$ *seq* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$

Siempre que los registros necesiten ser guardados, los generadores de código del compilador usan `preserving`, que es un método más sutil para combinar secuencias de instrucciones. `preserving` toma tres argumentos: un conjunto de registros y dos secuencias de instrucciones que deben ejecutarse secuencialmente. Anexa las secuencias de tal manera que el contenido de cada registro en el conjunto se preserve durante la ejecución de la primera secuencia, si esto es necesario para la ejecución de la segunda secuencia. Es decir, si la primera secuencia modifica el registro y la segunda secuencia realmente necesita el contenido original del registro, entonces `preserving` envuelve un `save` y un `restore` del registro alrededor de la primera secuencia antes de anexar las secuencias. De lo contrario, `preserving` simplemente devuelve las secuencias de instrucciones anexadas. Así, por ejemplo,

(preserving (list $\langle$ *reg* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$ $\langle$ *reg* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$) $\langle$ *seq* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 1}}\rangle$ $\langle$ *seq* $\color{SchemeDark}_{\hbox{\ttfamily\scriptsize 2}}\rangle$)

produce una de las siguientes cuatro secuencias de instrucciones, dependiendo de cómo ⟨seq_1⟩ y ⟨seq_2⟩ usan ⟨reg_1⟩ y ⟨reg_2⟩:

$$\vbox{ \offinterlineskip \halign{ \strut \kern0.8em # \kern0.4em \hfil & \vrule \kern0.8em # \kern0.4em \hfil & \vrule \kern0.8em # \kern0.4em \hfil & \vrule \kern0.8em # \kern0.4em \hfil \cr $\langle{\mathit{seq}_1}\rangle$ & \hbox{\tt (save} $\langle{\mathit{reg}_1}\rangle${\tt)} & \hbox{\tt (save} $\langle{\mathit{reg}_2}\rangle${\tt)} & \hbox{\tt (save} $\langle{\mathit{reg}_2}\rangle${\tt)} \cr $\langle{\mathit{seq}_2}\rangle$ & $\langle{\mathit{seq}_1}\rangle$ & $\langle{\mathit{seq}_1}\rangle$ & \hbox{\tt (save} $\langle{\mathit{reg}_1}\rangle${\tt)} \cr & \hbox{\tt (restore} $\langle{\mathit{reg}_1}\rangle${\tt)} & \hbox{\tt (restore} $\langle{\mathit{reg}_2}\rangle${\tt)} & $\langle{\mathit{seq}_1}\rangle$ \cr & $\langle{\mathit{seq}_2}\rangle$ & $\langle{\mathit{seq}_2}\rangle$ & \hbox{\tt (restore} $\langle{\mathit{reg}_1}\rangle${\tt)} \cr & & & \hbox{\tt (restore} $\langle{\mathit{reg}_2}\rangle${\tt)} \cr & & & $\langle{\mathit{seq}_2}\rangle$ \cr } }$$

Al usar `preserving` para combinar secuencias de instrucciones, el compilador evita operaciones de pila innecesarias. Esto también aísla los detalles de si generar o no instrucciones `save` y `restore` dentro del

procedimiento `preserving`, separándolos de las preocupaciones que surgen al escribir cada uno de los generadores de código individuales. De hecho, ninguna instrucción `save` o `restore` es producida explícitamente por los generadores de código.

En principio, podríamos representar una secuencia de instrucciones simplemente como una lista de instrucciones. `append/instruction/sequences` podría entonces combinar secuencias de instrucciones realizando un `append` de lista ordinario. Sin embargo, `preserving` sería entonces una operación compleja, porque tendría que analizar cada secuencia de instrucciones para determinar cómo la secuencia usa sus registros. `preserving` sería ineficiente además de complejo, porque tendría que analizar cada uno de sus argumentos de secuencia de instrucciones, aunque estas secuencias podrían haber sido construidas por llamadas a `preserving`, en cuyo caso sus partes ya habrían sido analizadas. Para evitar tal análisis repetitivo, asociaremos con cada instrucción secuencia alguna información sobre su uso de registros. Cuando construimos una secuencia de instrucciones básica, proporcionaremos esta información explícitamente, y los procedimientos que combinan secuencias de instrucciones derivarán información de uso de registros para la secuencia combinada a partir de la información asociada con las secuencias componentes.

Una secuencia de instrucciones contendrá tres piezas de información:

- el conjunto de registros que deben inicializarse antes de que se ejecuten las instrucciones en la secuencia (se dice que estos registros son *necesarios* por la secuencia),
- el conjunto de registros cuyos valores son modificados por las instrucciones en la secuencia, y
- las instrucciones reales (también llamadas *sentencias*) en la secuencia.

Representaremos una secuencia de instrucciones como una lista de sus tres partes. El constructor para secuencias de instrucciones es, por lo tanto

(define (make-instruction-sequence needs modifies statements) (list needs modifies statements))

Por ejemplo, la secuencia de dos instrucciones que busca el valor de la variable `x` en el entorno actual, asigna el resultado a `val`, y luego regresa, requiere que los registros `env` y `continue` hayan sido inicializados, y modifica el registro `val`. Esta secuencia, por lo tanto, se construiría como

(make-instruction-sequence '(env continue) '(val) '((assign val (op lookup-

variable-value) (const x) (reg env)) (goto (reg continue))))

A veces necesitamos construir una secuencia de instrucciones sin sentencias:

(define (empty-instruction-sequence) (make-instruction-sequence '() '() '()))

Los procedimientos para combinar secuencias de instrucciones se muestran en la Sección 5.5.4.

Ejercicio 5.31: Al evaluar una aplicación de procedimiento, el evaluador de control explícito siempre guarda y restaura el registro `env` alrededor de la evaluación del operador, guarda y restaura `env` alrededor de la evaluación de cada operando (excepto el último), guarda y restaura `argl` alrededor de la evaluación de cada operando, y guarda y restaura `proc` alrededor de la evaluación de la secuencia de operandos. Para cada una de las siguientes combinaciones, diga cuáles de estas operaciones `save` y `restore` son superfluas y, por lo tanto, podrían ser eliminadas por el mecanismo `preserving` del compilador:

(f 'x 'y) ((f) 'x 'y) (f (g 'x) y) (f (g 'x) 'y)

Ejercicio 5.32: Usando el mecanismo `preserving`, el compilador evitará guardar y restaurar `env` alrededor de la evaluación del operador de una combinación en el caso en que el operador es un símbolo. También podríamos incorporar tales optimizaciones en el evaluador. De hecho, el evaluador de control explícito de Section 5.4 ya realiza una optimización similar, tratando las combinaciones sin operandos como un caso especial.

1. Extienda el evaluador de control explícito para que reconozca como una clase separada de expresiones las combinaciones cuyo operador es un símbolo, y para aprovechar este hecho al evaluar tales expresiones.

2. Alyssa P. Hacker sugiere que al extender el evaluador para que reconozca más y más casos especiales podríamos incorporar todas las optimizaciones del compilador, y que esto eliminaría por completo la ventaja de la compilación. ¿Qué piensa de esta idea?

Compilando Expresiones

En esta sección y la siguiente implementamos los generadores de código a los que despacha el procedimiento `compile`.

Compilando código de enlace

En general, la salida de cada generador de código terminará con instrucciones—generadas por el procedimiento `compile/linkage`—que implementan el enlace requerido. Si el enlace es `return`, entonces debemos generar la instrucción `(goto (reg continue))`. Esto necesita el registro `continue` y no modifica ningún registro. Si el enlace es `next`, entonces no necesitamos incluir ninguna instrucción adicional. De lo contrario, el enlace es una etiqueta, y generamos un `goto` a esa etiqueta, una instrucción que no necesita ni modifica ningún registro.[36]

(define (compile-linkage linkage) (cond ((eq? linkage 'return) (make-instruction-sequence '(continue) '() '((goto (reg continue))))) ((eq? linkage 'next) (empty-instruction-sequence)) (else (make-instruction-sequence '() '() `((goto (label ,linkage)))))))

El código de enlace se anexa a una secuencia de instrucciones mediante `preserving` el registro `continue`, ya que un enlace `return` requerirá el registro `continue`: Si la secuencia de instrucciones dada modifica `continue` y el código de enlace lo necesita, `continue` será guardado y restaurado.

(define (end-with-linkage linkage instruction-sequence) (preserving '(continue) instruction-sequence (compile-linkage linkage)))

Compilando expresiones simples

Los generadores de código para expresiones autoevaluables, citas y variables construyen secuencias de instrucciones que asignan el valor requerido al registro de destino y luego proceden como se especifica en el descriptor de enlace.

(define (compile-self-evaluating exp target linkage) (end-with-linkage linkage (make-instruction-sequence '() (list target) ((assign ,target (const ,exp)))))) (define (compile-quoted exp target linkage) (end-with-linkage linkage (make-instruction-sequence '() (list target)((assign ,target (const ,(text-of-quotation exp))))))) (define (compile-variable exp target linkage) (end-with-linkage linkage (make-instruction-sequence '(env) (list target) `((assign ,target (op lookup-variable-value) (const ,exp) (reg env)))))

Todas estas instrucciones de asignación modifican el registro de destino, y la que busca una variable necesita el registro `env`. Las asignaciones y definiciones se manejan de manera muy similar a como se hace en el intérprete. Generamos recursivamente código que calcula el valor a ser asignado a la variable, y le anexamos una secuencia de dos instrucciones que realmente establece o define la variable y asigna el valor de la expresión completa (el símbolo `ok`) al registro de destino. La compilación recursiva

tiene como destino `val` y enlace `next` para que el código ponga su resultado en `val` y continúe con el código que se anexa después. El anexo se realiza preservando `env`, ya que el entorno es necesario para establecer o definir la variable y el código para el valor de la variable podría ser la compilación de una expresión compleja que podría modificar los registros de maneras arbitrarias.

(define (compile-assignment exp target linkage) (let ((var (assignment-variable exp)) (get-value-code (compile (assignment-value exp) 'val 'next))) (end-with-linkage linkage (preserving '(env) get-value-code (make-instruction-sequence '(env val) (list target) ((perform (op set-variable-value!) (const ,var) (reg val) (reg env)) (assign ,target (const ok)))))))) (define (compile-definition exp target linkage) (let ((var (definition-variable exp)) (get-value-code (compile (definition-value exp) 'val 'next))) (end-with-linkage linkage (preserving '(env) get-value-code (make-instruction-sequence '(env val) (list target)((perform (op define-variable!) (const ,var) (reg val) (reg env)) (assign ,target (const ok)))))))

La secuencia de dos instrucciones anexada requiere `env` y `val` y modifica el destino. Tenga en cuenta que aunque preservamos `env` para esta secuencia, no preservamos `val`, porque el `get/value/code` está diseñado para colocar explícitamente su resultado en `val` para que sea utilizado por esta secuencia. (De hecho, si preserváramos `val`, tendríamos un error, porque esto causaría que el contenido anterior de `val` se restaurara justo después de que se ejecute el `get/value/code`).

Compilando expresiones condicionales

El código para una expresión `if` compilada con un destino y enlace dados tiene la forma

< *compilación del predicado, destino* val*, enlace* next > (test (op false?) (reg val)) (branch (label false-branch)) true-branch < *compilación del consecuente con destino dado y enlace dado o* after-if > false-branch < *compilación de la alternativa con destino y enlace dados* > after-if

Para generar este código, compilamos el predicado, el consecuente y la alternativa, y combinamos el código resultante con instrucciones para probar el resultado del predicado y con etiquetas recién generadas para marcar las ramas verdadera y falsa y el final del condicional.[37] En esta disposición del código, debemos saltar alrededor de la rama verdadera si la prueba es falsa. La única complicación leve está en cómo se debe manejar el enlace para la rama verdadera. Si el enlace para el condicional es `return` o una etiqueta, entonces las ramas verdadera y falsa usarán este mismo enlace. Si el enlace

es `next`, la rama verdadera termina con un salto alrededor del código para la rama falsa a la etiqueta al final del condicional.

(define (compile-if exp target linkage) (let ((t-branch (make-label 'true-branch)) (f-branch (make-label 'false-branch)) (after-if (make-label 'after-if))) (let ((consequent-linkage (if (eq? linkage 'next) after-if linkage))) (let ((p-code (compile (if-predicate exp) 'val 'next)) (c-code (compile (if-consequent exp) target consequent-linkage)) (a-code (compile (if-alternative exp) target linkage))) (preserving '(env continue) p-code (append-instruction-sequences (make-instruction-sequence '(val) '() '((test (op false?) (reg val)) (branch (label ,f-branch)))) (parallel-instruction-sequences (append-instruction-sequences t-branch c-code) (append-instruction-sequences f-branch a-code)) after-if))))))

`env` se preserva alrededor del código del predicado porque podría ser necesario para las ramas verdadera y falsa, y `continue` se preserva porque podría ser necesario para el código de enlace en esas ramas. El código para las ramas verdadera y falsa (que no se ejecutan secuencialmente) se anexa usando un combinador especial `parallel/instruction/sequences` descrito en la Sección 5.5.4. Tenga en cuenta que `cond` es una expresión derivada, por lo que todo lo que el compilador necesita hacer para manejarla es aplicar el transformador `cond/>if` (de Section 4.1.2) y compilar la expresión `if` resultante.

Compilando secuencias

La compilación de secuencias (de cuerpos de procedimientos o expresiones explícitas `begin`) es paralela a su evaluación. Cada expresión de la secuencia se compila—la última expresión con el enlace especificado para la secuencia, y las otras expresiones con enlace `next` (para ejecutar el resto de la secuencia). Las secuencias de instrucciones para las expresiones individuales se anexan para formar una única secuencia de instrucciones, de modo que `env` (necesario para el resto de la secuencia) y `continue` (posiblemente necesario para el enlace al final de la secuencia) se preserven.

(define (compile-sequence seq target linkage) (if (last-exp? seq) (compile (first-exp seq) target linkage) (preserving '(env continue) (compile (first-exp seq) target 'next) (compile-sequence (rest-exps seq) target linkage))))

Compilando expresiones `lambda`

Las expresiones `lambda` construyen procedimientos. El código objeto para una expresión `lambda` debe tener la forma

⟨ *construir objeto de procedimiento y asignarlo al registro de destino* ⟩
⟨ *enlace* ⟩

Cuando compilamos la expresión `lambda`, también generamos el código para el cuerpo del procedimiento. Aunque el cuerpo no se ejecutará en el momento de la construcción del procedimiento, es conveniente insertarlo en el código objeto justo después del código para el `lambda`. Si el enlace para la expresión `lambda` es una etiqueta o `return`, esto está bien. Pero si el enlace es `next`, necesitaremos saltar alrededor del código para el cuerpo del procedimiento usando un enlace que salta a una etiqueta que se inserta después del cuerpo. El código objeto, por lo tanto, tiene la forma

⟨ *construir objeto de procedimiento y asignarlo al registro de destino* ⟩
⟨ *código para el enlace dado* ⟩ *o* `(goto (label after/lambda))`
⟨ *compilación del cuerpo del procedimiento* ⟩ after-lambda

`compile/lambda` genera el código para construir el objeto de procedimiento seguido por el código para el cuerpo del procedimiento. El objeto de procedimiento será construido en tiempo de ejecución combinando el entorno actual (el entorno en el punto de definición) con el punto de entrada al cuerpo del procedimiento compilado (una etiqueta recién generada).[38]

(define (compile-lambda exp target linkage) (let ((proc-entry (make-label 'entry)) (after-lambda (make-label 'after-lambda))) (let ((lambda-linkage (if (eq? linkage 'next) after-lambda linkage))) (append-instruction-sequences (tack-on-instruction-sequence (end-with-linkage lambda-linkage (make-instruction-sequence '(env) (list target) '((assign ,target (op make-compiled-procedure) (label ,proc-entry) (reg env))))) (compile-lambda-body exp proc-entry)) after-lambda))))

`compile/lambda` utiliza el combinador especial `tack/on/instruction/sequence` en lugar de `append/instruction/sequences` (Section 5.5.4) para anexar el cuerpo del procedimiento al código de la expresión `lambda`, porque el cuerpo no es parte de la secuencia de instrucciones que se ejecutará cuando se ingrese a la secuencia combinada; más bien, está en la secuencia solo porque ese era un lugar conveniente para ponerlo. `compile/lambda/body` construye el código para el cuerpo del procedimiento. Este código comienza con una etiqueta para el punto de entrada. Luego vienen las instrucciones que harán que el entorno de evaluación en tiempo de ejecución cambie al entorno correcto para evaluar el cuerpo del procedimiento, a saber, el entorno de definición del procedimiento, extendido para incluir las vinculaciones de los parámetros formales a los argumentos con los que se llama al procedimiento. Después de esto viene el código para la secuencia de expresiones que conforma el cuerpo del procedimiento. La secuencia se

compila con enlace `return` y destino `val` para que termine regresando del procedimiento con el resultado del procedimiento en `val`.

(define (compile-lambda-body exp proc-entry) (let ((formals (lambda-parameters exp))) (append-instruction-sequences (make-instruction-sequence '(env proc argl) '(env) `(,proc-entry (assign env (op compiled-procedure-env) (reg proc)) (assign env (op extend-environment) (const ,formals) (reg argl) (reg env)))) (compile-sequence (lambda-body exp) 'val 'return))))

Compilando Combinaciones

La esencia del proceso de compilación es la compilación de aplicaciones de procedimientos. El código para una combinación compilada con un destino y enlace dados tiene la forma

⟨ *compilación del operador, destino* proc*, enlace* next ⟩ ⟨ *evaluar operandos y construir lista de argumentos en* argl ⟩ ⟨ *compilación de la llamada al procedimiento con el destino y enlace dados* ⟩

Los registros `env`, `proc` y `argl` pueden tener que ser guardados y restaurados durante la evaluación del operador y los operandos. Tenga en cuenta que este es el único lugar en el compilador donde se especifica un destino que no sea `val`. El código requerido es generado por `compile/application`. Esto compila recursivamente el operador, para producir código que pone el procedimiento a aplicar en `proc`, y compila los operandos, para producir código que evalúa los operandos individuales de la aplicación. Las secuencias de instrucciones para los operandos se combinan (por `construct/arglist`) con código que construye la lista de argumentos en `argl`, y el código de la lista de argumentos resultante se combina con el código del procedimiento y el código que realiza la llamada al procedimiento (producido por `compile/procedure/call`). Al anexar las secuencias de código, el registro `env` debe preservarse alrededor de la evaluación del operador (ya que la evaluación del operador podría modificar `env`, que será necesario para evaluar los operandos), y el registro `proc` debe preservarse alrededor de la construcción de la lista de argumentos (ya que la evaluación de los operandos podría modificar `proc`, que será necesario para la aplicación real del procedimiento). `continue` también debe preservarse en todo momento, ya que es necesario para el enlace en la llamada al procedimiento.

(define (compile-application exp target linkage) (let ((proc-code (compile (operator exp) 'proc 'next)) (operand-codes (map (lambda (operand) (compile operand 'val 'next)) (operands exp)))) (preserving '(env continue) proc-code (preserving '(proc continue) (construct-arglist operand-codes) (compile-procedure-call target linkage)))))

El código para construir la lista de argumentos evaluará cada operando en `val` y luego hará `cons` de ese valor en la lista de argumentos que se acumula en `argl`. Como hacemos `cons` de los argumentos en `argl` en secuencia, debemos comenzar con el último argumento y terminar con el primero, para que los argumentos aparezcan en orden del primero al último en la lista resultante. En lugar de desperdiciar una instrucción inicializando `argl` a la lista vacía para prepararse para esta secuencia de evaluaciones, hacemos que la primera secuencia de código construya el `argl` inicial. La forma general de la construcción de la lista de argumentos es, por lo tanto, la siguiente:

⟨ *compilación del último operando, con destino a* val ⟩ (assign argl (op list) (reg val)) ⟨ *compilación del siguiente operando, con destino a* val ⟩ (assign argl (op cons) (reg val) (reg argl)) … ⟨ *compilación del primer operando, con destino a* val ⟩ (assign argl (op cons) (reg val) (reg argl))

`argl` debe preservarse alrededor de cada evaluación de operando excepto la primera (para que los argumentos acumulados hasta ahora no se pierdan), y `env` debe preservarse alrededor de cada evaluación de operando excepto la última (para que lo usen las evaluaciones de operandos posteriores). Compilar este código de argumento es un poco complicado, debido al tratamiento especial del primer operando a ser evaluado y la necesidad de preservar `argl` y `env` en diferentes lugares. El procedimiento `construct/arglist` toma como argumentos el código que evalúa los operandos individuales. Si no hay operandos en absoluto, simplemente emite la instrucción

(assign argl (const ()))

De lo contrario, `construct/arglist` crea código que inicializa `argl` con el último argumento, y anexa código que evalúa el resto de los argumentos y los adjunta a `argl` en sucesión. Para procesar los argumentos del último al primero, debemos invertir la lista de secuencias de código de operandos del orden proporcionado por `compile/application`.

(define (construct-arglist operand-codes) (let ((operand-codes (reverse operand-codes))) (if (null? operand-codes) (make-instruction-sequence '() '(argl) '((assign argl (const ()))))) (let ((code-to-get-last-arg (append-instruction-sequences (car operand-codes) (make-instruction-sequence '(val) '(argl) '((assign argl (op list) (reg val)))))))) (if (null? (cdr operand-codes)) code-to-get-last-arg (preserving '(env) code-to-get-last-arg (code-to-get-rest-args (cdr operand-codes)))))))) (define (code-to-get-rest-args operand-codes) (let ((code-for-next-arg (preserving '(argl) (car operand-codes) (make-instruction-sequence '(val argl) '(argl) '((assign argl (op cons) (reg val) (reg argl))))))) (if (null? (cdr operand-codes)) code-for-next-arg (preserving '(env)

code-for-next-arg (code-to-get-rest-args (cdr operand-codes))))))

Aplicando procedimientos

Después de evaluar los elementos de una combinación, el código compilado debe aplicar el procedimiento en `proc` a los argumentos en `argl`. El código realiza esencialmente el mismo despacho que el procedimiento `apply` en el evaluador metacircular de Section 4.1.1 o el punto de entrada `apply/dispatch` en el evaluador de control explícito de Section 5.4.1. Comprueba si el procedimiento a aplicar es un procedimiento primitivo o un procedimiento compilado. Para un procedimiento primitivo, utiliza `apply/primitive/procedure`; veremos en breve cómo maneja los procedimientos compilados. El código de aplicación de procedimiento tiene la siguiente forma:

(test (op primitive-procedure?) (reg proc)) (branch (label primitive-branch)) compiled-branch ⟨ *código para aplicar el procedimiento compilado con el destino dado y el enlace apropiado* ⟩ primitive-branch (assign ⟨ *target* ⟩ (op apply-primitive-procedure) (reg proc) (reg argl)) ⟨ *enlace* ⟩ after-call

Observe que la rama compilada debe saltar alrededor de la rama primitiva. Por lo tanto, si el enlace para la llamada al procedimiento original era `next`, la rama compuesta debe usar un enlace que salte a una etiqueta que se inserta después de la rama primitiva. (Esto es similar al enlace utilizado para la rama verdadera en `compile/if`).

(define (compile-procedure-call target linkage) (let ((primitive-branch (make-label 'primitive-branch)) (compiled-branch (make-label 'compiled-branch)) (after-call (make-label 'after-call))) Generated code (let ((compiled-linkage (if (eq? linkage 'next) after-call linkage))) (append-instruction-sequences (make-instruction-sequence '(proc) '() `((test (op primitive-procedure?) (reg proc)) (branch (label ,primitive-branch)))) (parallel-instruction-sequences (append-instruction-sequences compiled-branch (compile-proc-appl target compiled-linkage)) (append-instruction-sequences primitive-branch (end-with-linkage linkage (make-instruction-sequence '(proc argl) (list target) `((assign ,target (op apply-primitive-procedure) (reg proc) (reg argl))))))) after-call)))) Use code with caution.

Las ramas primitiva y compuesta, al igual que las ramas verdadera y falsa en `compile/if`, se anexan usando `parallel/instruction/sequences` en lugar del ordinario `append/instruction/sequences`, porque no serán ejecutadas secuencialmente.

Aplicando procedimientos compilados

El código que maneja la aplicación de procedimientos es la parte más sutil del compilador, aunque las secuencias de instrucciones que genera son muy cortas. Un procedimiento compilado (construido por `compile/lambda`) tiene un punto de entrada, que es una etiqueta que designa dónde comienza el código para el procedimiento. El código en este punto de entrada calcula un resultado en `val` y regresa ejecutando la instrucción `(goto (reg continue))`. Por lo tanto, podríamos esperar que el código para una aplicación de procedimiento compilado (a ser generado por `compile/proc/appl`) con un destino y enlace dados se vea así si el enlace es una etiqueta

(assign continue (label proc-return)) (assign val (op compiled-procedure-entry) (reg proc)) (goto (reg val)) proc-return (assign ⟨ *target* ⟩ (reg val)) ; incluido si el destino no es `val` (goto (label ⟨ *linkage* ⟩)) ; código de enlace

o así si el enlace es `return`.

(save continue) (assign continue (label proc-return)) (assign val (op compiled-procedure-entry) (reg proc)) (goto (reg val)) proc-return (assign ⟨ *target* ⟩ (reg val)) ; incluido si el destino no es `val` (restore continue) (goto (reg continue)) ; código de enlace

Este código configura `continue` para que el procedimiento regrese a una etiqueta `proc/return` y salta al punto de entrada del procedimiento. El código en `proc/return` transfiere el resultado del procedimiento de `val` al registro de destino (si es necesario) y luego salta a la ubicación especificada por el enlace. (El enlace es siempre `return` o una etiqueta, porque `compile/procedure/call` reemplaza un enlace `next` para la rama de procedimiento compuesto por una etiqueta `after/call`). De hecho, si el destino no es `val`, ese es exactamente el código que nuestro compilador generará.[39] Sin embargo, por lo general, el destino es `val` (la única vez que el compilador especifica un registro diferente es cuando se dirige la evaluación de un operador a `proc`), por lo que el resultado del procedimiento se pone directamente en el registro de destino y no hay necesidad de regresar a una ubicación especial que lo copie. En su lugar, simplificamos el código configurando `continue` para que el procedimiento "regrese" directamente al lugar especificado por el enlace del llamador:

⟨ *configurar* continue *para el enlace* ⟩ (assign val (op compiled-procedure-entry) (reg proc)) (goto (reg val))

Si el enlace es una etiqueta, configuramos `continue` para que el procedimiento regrese a esa etiqueta. (Es decir, el `(goto (reg continue))` con el que termina el procedimiento se vuelve equivalente al `(goto (label`

⟨*linkage*⟩)) en `proc/return` arriba).

(assign continue (label ⟨ *linkage* ⟩)) (assign val (op compiled-procedure-entry) (reg proc)) (goto (reg val))

Si el enlace es `return`, no necesitamos configurar `continue` en absoluto: ya contiene la ubicación deseada. (Es decir, el `(goto (reg continue))` con el que termina el procedimiento va directamente al lugar donde el `(goto (reg continue))` en `proc/return` habría ido).

(assign val (op compiled-procedure-entry) (reg proc)) (goto (reg val))

Con esta implementación del enlace `return`, el compilador genera código recursivo de cola. Llamar a un procedimiento como el paso final en el cuerpo de un procedimiento hace una transferencia directa, sin guardar ninguna información en la pila. Supongamos, en cambio, que hubiéramos manejado el caso de una llamada a procedimiento con un enlace de `return` y un destino de `val` como se muestra arriba para un destino que no es `val`. Esto destruiría la recursión de cola. Nuestro sistema seguiría dando el mismo valor para cualquier expresión. Pero cada vez que llamáramos a un procedimiento, guardaríamos `continue` y regresaríamos después de la llamada para deshacer el guardado (inútil). Estos guardados adicionales se acumularían durante un nido de llamadas a procedimientos. [40] `compile/proc/appl` genera el código de aplicación de procedimiento anterior considerando cuatro casos, dependiendo de si el destino para la llamada es `val` y si el enlace es `return`. Observe que las secuencias de instrucciones se declaran para modificar todos los registros, ya que la ejecución del cuerpo del procedimiento puede cambiar los registros de formas arbitrarias.[41] También tenga en cuenta que la secuencia de código para el caso con destino `val` y enlace `return` se declara que necesita `continue`: aunque `continue` no se usa explícitamente en la secuencia de dos instrucciones, debemos asegurarnos de que `continue` tendrá el valor correcto cuando entremos al procedimiento compilado.

(define (compile-proc-appl target linkage) (cond ((and (eq? target 'val) (not (eq? linkage 'return))) (make-instruction-sequence '(proc) all-regs ((assign continue (label ,linkage)) (assign val (op compiled-procedure-entry) (reg proc)) (goto (reg val))))) ((and (not (eq? target 'val)) (not (eq? linkage 'return))) (let ((proc-return (make-label 'proc-return))) (make-instruction-sequence '(proc) all-regs((assign continue (label ,proc-return)) (assign val (op compiled-procedure-entry) (reg proc)) (goto (reg val)) ,proc-return (assign ,target (reg val)) (goto (label ,linkage)))))) ((and (eq? target 'val) (eq? linkage 'return)) (make-instruction-sequence '(proc continue) all-regs '((assign val (op compiled-procedure-entry) (reg proc)) (goto (reg val))))) ((and (not (eq? target 'val)) (eq? linkage 'return)) (error "return linkage,

target not val: COMPILE" target))))

Combinando Secuencias de Instrucciones

Esta sección describe los detalles sobre cómo se representan y combinan las secuencias de instrucciones. Recordemos de Section 5.5.1 que una secuencia de instrucciones se representa como una lista de los registros necesarios, los registros modificados y las instrucciones reales. También consideraremos una etiqueta (símbolo) como un caso degenerado de una secuencia de instrucciones, que no necesita ni modifica ningún registro. Por lo tanto, para determinar los registros necesarios y modificados por las secuencias de instrucciones, usamos los selectores

(define (registers-needed s) (if (symbol? s) '() (car s))) (define (registers-modified s) (if (symbol? s) '() (cadr s))) (define (statements s) (if (symbol? s) (list s) (caddr s)))

y para determinar si una secuencia dada necesita o modifica un registro dado usamos los predicados

(define (needs-register? seq reg) (memq reg (registers-needed seq))) (define (modifies-register? seq reg) (memq reg (registers-modified seq)))

En términos de estos predicados y selectores, podemos implementar los diversos combinadores de secuencias de instrucciones utilizados en todo el compilador. El combinador básico es `append/instruction/sequences`. Este toma como argumentos un número arbitrario de secuencias de instrucciones que deben ejecutarse secuencialmente y devuelve una secuencia de instrucciones cuyas sentencias son las sentencias de todas las secuencias anexadas. El punto sutil es determinar los registros que son necesarios y modificados por la secuencia resultante. Modifica aquellos registros que son modificados por cualquiera de las secuencias; necesita aquellos registros que deben inicializarse antes de que la primera secuencia pueda ejecutarse (los registros necesarios para la primera secuencia), junto con aquellos registros necesarios para cualquiera de las otras secuencias que no son inicializados (modificados) por las secuencias que le preceden. Las secuencias se anexan de dos en dos por `append/2/sequences`. Esta toma dos secuencias de instrucciones `seq1` y `seq2` y devuelve la secuencia de instrucciones cuyas sentencias son las sentencias de `seq1` seguidas por las sentencias de `seq2`, cuyos registros modificados son aquellos registros que son modificados por `seq1` o `seq2`, y cuyos registros necesarios son los registros necesarios para `seq1` junto con aquellos registros necesarios para `seq2` que no son modificados por `seq1`. (En términos de operaciones de conjuntos, el nuevo conjunto de registros necesarios es la unión del conjunto de registros

necesarios para `seq1` con la diferencia de conjuntos de los registros necesarios para `seq2` y los registros modificados por `seq1`). Por lo tanto, `append/instruction/sequences` se implementa de la siguiente manera:

(define (append-instruction-sequences . seqs) (define (append-2-sequences seq1 seq2) (make-instruction-sequence (list-union (registers-needed seq1) (list-difference (registers-needed seq2) (registers-modified seq1))) (list-union (registers-modified seq1) (registers-modified seq2)) (append (statements seq1) (statements seq2)))) (define (append-seq-list seqs) (if (null? seqs) (empty-instruction-sequence) (append-2-sequences (car seqs) (append-seq-list (cdr seqs))))) (append-seq-list seqs))

Este procedimiento utiliza algunas operaciones simples para manipular conjuntos representados como listas, similar a la representación de conjuntos (no ordenados) descrita en Section 2.3.3:

(define (list-union s1 s2) (cond ((null? s1) s2) ((memq (car s1) s2) (list-union (cdr s1) s2)) (else (cons (car s1) (list-union (cdr s1) s2))))) (define (list-difference s1 s2) (cond ((null? s1) '()) ((memq (car s1) s2) (list-difference (cdr s1) s2)) (else (cons (car s1) (list-difference (cdr s1) s2)))))

`preserving`, el segundo combinador principal de secuencias de instrucciones, toma una lista de registros `regs` y dos secuencias de instrucciones `seq1` y `seq2` que deben ejecutarse secuencialmente. Devuelve una instrucción secuencia cuyas sentencias son las sentencias de `seq1` seguidas por las sentencias de `seq2`, con las instrucciones `save` y `restore` apropiadas alrededor de `seq1` para proteger los registros en `regs` que son modificados por `seq1` pero necesarios para `seq2`. Para lograr esto, `preserving` primero crea una secuencia que tiene los `save`s requeridos seguidos por las sentencias de `seq1` seguidas por los `restore`s requeridos. Esta secuencia necesita los registros que se guardan y restauran además de los registros necesarios para `seq1`, y modifica los registros modificados por `seq1` excepto los que se guardan y restauran. Esta secuencia aumentada y `seq2` se anexan luego de la manera habitual. El siguiente procedimiento implementa esta estrategia recursivamente, recorriendo la lista de registros a preservar:[42]

(define (preserving regs seq1 seq2) (if (null? regs) (append-instruction-sequences seq1 seq2) (let ((first-reg (car regs))) (if (and (needs-register? seq2 first-reg) (modifies-register? seq1 first-reg)) (preserving (cdr regs) (make-instruction-sequence (list-union (list first-reg) (registers-needed seq1)) (list-difference (registers-modified seq1) (list first-reg)) (append ((save ,first-reg)) (statements seq1)((restore ,first-reg)))) seq2) (preserving (cdr regs) seq1 seq2)))))

Otro combinador de secuencias, `tack/on/instruction/sequence`, es utilizado por `compile/lambda` para anexar un cuerpo de procedimiento a otra secuencia. Debido a que el cuerpo del procedimiento no está "en línea" para ser ejecutado como parte de la secuencia combinada, su uso de registros no tiene impacto en el uso de registros de la secuencia en la que está incrustado. Por lo tanto, ignoramos los conjuntos de registros necesarios y modificados del cuerpo del procedimiento cuando lo adjuntamos a la otra secuencia.

(define (tack-on-instruction-sequence seq body-seq) (make-instruction-sequence (registers-needed seq) (registers-modified seq) (append (statements seq) (statements body-seq))))

`compile/if` y `compile/procedure/call` usan un combinador especial llamado `parallel/instruction/sequences` para anexar las dos ramas alternativas que siguen a una prueba. Las dos ramas nunca se ejecutarán secuencialmente; para cualquier evaluación particular de la prueba, se entrará en una rama u otra. Debido a esto, los registros necesarios para la segunda rama siguen siendo necesarios para la secuencia combinada, incluso si estos son modificados por la primera rama.

(define (parallel-instruction-sequences seq1 seq2) (make-instruction-sequence (list-union (registers-needed seq1) (registers-needed seq2)) (list-union (registers-modified seq1) (registers-modified seq2)) (append (statements seq1) (statements seq2))))

Un Ejemplo de Código Compilado

Ahora que hemos visto todos los elementos del compilador, examinemos un ejemplo de código compilado para ver cómo encajan las cosas. Compilaremos la definición de un procedimiento `factorial` recursivo llamando a `compile`:

(compile '(define (factorial n) (if (= n 1) 1 (* (factorial (- n 1)) n))) 'val 'next)

Hemos especificado que el valor de la expresión `define` debe ser colocado en el registro `val`. No nos importa lo que haga el código compilado después de ejecutar el `define`, por lo que nuestra elección de `next` como descriptor de enlace es arbitraria. `compile` determina que la expresión es una definición, por lo que llama a `compile/definition` para compilar código para calcular el valor a asignar (dirigido a `val`), seguido de código para instalar la definición, seguido por código para poner el valor del `define` (que es el símbolo `ok`) en el registro de destino, seguido finalmente por el código de enlace. `env` se preserva alrededor del cálculo del valor, porque es necesario para instalar la definición. Debido a que el enlace es `next`, no hay código de

enlace en este caso. El esqueleto del código compilado es, por lo tanto

⟨ *guardar* env *si es modificado por el código para calcular el valor* ⟩ ⟨ *compilación del valor de definición, destino* val*, enlace* next ⟩ ⟨ *restaurar* env *si se guardó arriba* ⟩ (perform (op define-variable!) (const factorial) (reg val) (reg env)) (assign val (const ok))

La expresión que debe compilarse para producir el valor para la variable `factorial` es una expresión `lambda` cuyo valor es el procedimiento que calcula factoriales. `compile` maneja esto llamando a `compile/lambda`, que compila el cuerpo del procedimiento, lo etiqueta como un nuevo punto de entrada, y genera la instrucción que combinará el cuerpo del procedimiento en el nuevo punto de entrada con el entorno de tiempo de ejecución y asignará el resultado a `val`. La secuencia luego salta alrededor del código del procedimiento compilado, que se inserta en este punto. El código del procedimiento en sí comienza extendiendo el entorno de definición del procedimiento con un marco que vincula el parámetro formal `n` al argumento del procedimiento. Luego viene el cuerpo real del procedimiento. Como este código para el valor de la variable no modifica el registro `env`, los `save` y `restore` opcionales mostrados arriba no se generan. (El código del procedimiento en `entry2` no se ejecuta en este punto, por lo que su uso de `env` es irrelevante.) Por lo tanto, el esqueleto del código compilado se convierte en

(assign val (op make-compiled-procedure) (label entry2) (reg env)) (goto (label after-lambda1)) entry2 (assign env (op compiled-procedure-env) (reg proc)) (assign env (op extend-environment) (const (n)) (reg argl) (reg env)) ⟨ *compilación del cuerpo del procedimiento* ⟩ after-lambda1 (perform (op define-variable!) (const factorial) (reg val) (reg env)) (assign val (const ok))

El cuerpo de un procedimiento siempre se compila (por `compile/lambda/body`) como una secuencia con destino `val` y enlace `return`. La secuencia en este caso consiste en una sola expresión `if`:

(if (= n 1) 1 (* (factorial (- n 1)) n))

`compile/if` genera código que primero calcula el predicado (dirigido a `val`), luego verifica el resultado y salta alrededor de la rama verdadera si el predicado es falso. `env` y `continue` se preservan alrededor del código del predicado, ya que pueden ser necesarios para el resto de la expresión `if`. Dado que la expresión `if` es la expresión final (y única expresión) en la secuencia que conforma el cuerpo del procedimiento, su destino es `val` y su enlace es `return`, por lo que las ramas verdadera y falsa se compilan ambas con destino `val` y enlace `return`. (Es decir, el valor del condicional, que es el

valor calculado por cualquiera de sus ramas, es el valor del procedimiento).

< *guardar* continue*,* env *si es modificado por el predicado y necesario para las ramas* > < *compilación del predicado, destino* val*, enlace* next > < *restaurar* continue*,* env *si se guardó arriba* > (test (op false?) (reg val)) (branch (label false-branch4)) true-branch5 < *compilación de la rama verdadera, destino* val*, enlace* return > false-branch4 < *compilación de la rama falsa, destino* val*, enlace* return > after-if3

El predicado `(= n 1)` es una llamada a procedimiento. Esto busca el operador (el símbolo `=`) y coloca este valor en `proc`. Luego ensambla los argumentos `1` y el valor de `n` en `argl`. Luego prueba si `proc` contiene un procedimiento primitivo o compuesto, y despacha a una rama primitiva o una rama compuesta en consecuencia. Ambas ramas se reanudan en la etiqueta `after/call`. Los requisitos para preservar registros alrededor de la evaluación del operador y los operandos no resultan en ningún guardado de registros, porque en este caso esas evaluaciones no modifican los registros en cuestión.

(assign proc (op lookup-variable-value) (const =) (reg env)) (assign val (const 1)) (assign argl (op list) (reg val)) (assign val (op lookup-variable-value) (const n) (reg env)) (assign argl (op cons) (reg val) (reg argl)) (test (op primitive-procedure?) (reg proc)) (branch (label primitive-branch17)) compiled-branch16 (assign continue (label after-call15)) (assign val (op compiled-procedure-entry) (reg proc)) (goto (reg val)) primitive-branch17 (assign val (op apply-primitive-procedure) (reg proc) (reg argl)) after-call15

La rama verdadera, que es la constante 1, se compila (con destino `val` y enlace `return`) a

(assign val (const 1)) (goto (reg continue))

El código para la rama falsa es otra llamada a procedimiento, donde el procedimiento es el valor del símbolo `*`, y los argumentos son `n` y el resultado de otra llamada a procedimiento (una llamada a `factorial`). Cada una de estas llamadas configura `proc` y `argl` y sus propias ramas primitiva y compuesta. La Figura 5.17 muestra la compilación completa de la definición del procedimiento `factorial`. Observe que los posibles `save` y `restore` de `continue` y `env` alrededor del predicado, mostrados anteriormente, se generan de hecho, porque estos registros son modificados por la llamada a procedimiento en el predicado y necesarios para la llamada a procedimiento y el enlace `return` en las ramas.

Ejercicio 5.33: Considere la siguiente definición de un procedimiento factorial, que es ligeramente diferente de la dada anteriormente:

(define (factorial-alt n) (if (= n 1) 1 (* n (factorial-alt (- n 1)))))

Compile este procedimiento y compare el código resultante con el producido para `factorial`. Explique cualquier diferencia que encuentre. ¿Alguno de los programas se ejecuta más eficientemente que el otro?

Ejercicio 5.34: Compile el procedimiento factorial iterativo

(define (factorial n) (define (iter product counter) (if (> counter n) product (iter (* counter product) (+ counter 1)))) (iter 1 1))

Anote el código resultante, mostrando la diferencia esencial entre el código para las versiones iterativa y recursiva de `factorial` que hace que un proceso acumule espacio en la pila y el otro se ejecute en espacio de pila constante.

Figura 5.17: ↓ Compilación de la definición del procedimiento `factorial`.

;; construir el procedimiento y saltar sobre el código del cuerpo del procedimiento (assign val (op make-compiled-procedure) (label entry2) (reg env)) (goto (label after-lambda1)) entry2 ;; las llamadas a `factorial` entrarán aquí (assign env (op compiled-procedure-env) (reg proc)) (assign env (op extend-environment) (const (n)) (reg argl) (reg env)) ;; comenzar el cuerpo real del procedimiento (save continue) (save env) ;; calcular `(= n 1)` (assign proc (op lookup-variable-value) (const =) (reg env)) (assign val (const 1)) (assign argl (op list) (reg val)) (assign val (op lookup-variable-value) (const n) (reg env)) (assign argl (op cons) (reg val) (reg argl)) (test (op primitive-procedure?) (reg proc)) (branch (label primitive-branch17)) compiled-branch16 (assign continue (label after-call15)) (assign val (op compiled-procedure-entry) (reg proc)) (goto (reg val)) primitive-branch17 (assign val (op apply-primitive-procedure) (reg proc) (reg argl)) after-call15 ;; `val` ahora contiene el resultado de `(= n 1)` (restore env) (restore continue) (test (op false?) (reg val)) (branch (label false-branch4)) true-branch5 ;; devolver 1 (assign val (const 1)) (goto (reg continue)) false-branch4 ;; calcular y devolver `(* (factorial (- n 1)) n)` (assign proc (op lookup-variable-value) (const) (reg env)) (save continue) (save proc) ;; guardar procedimiento (assign val (op lookup-variable-value) (const n) (reg env)) (assign argl (op list) (reg val)) (save argl) ;; guardar lista de argumentos parcial para ;; calcular `(factorial (- n 1))`, que es el otro argumento para (assign proc (op lookup-variable-value) (const factorial) (reg env)) (save proc) ;; guardar procedimiento `factorial` ;; calcular `(- n 1)`, que es el argumento para `factorial` (assign proc (op lookup-variable-value)

(const -) (reg env)) (assign val (const 1)) (assign argl (op list) (reg val)) (assign val (op lookup-variable-value) (const n) (reg env)) (assign argl (op cons) (reg val) (reg argl)) (test (op primitive-procedure?) (reg proc)) (branch (label primitive-branch8)) compiled-branch7 (assign continue (label after-call6)) (assign val (op compiled-procedure-entry) (reg proc)) (goto (reg val)) primitive-branch8 (assign val (op apply-primitive-procedure) (reg proc) (reg argl)) after-call6 ;; `val` ahora contiene el resultado de `(- n 1)` (assign argl (op list) (reg val)) (restore proc) ;; restaurar `factorial` ;; aplicar `factorial` (test (op primitive-procedure?) (reg proc)) (branch (label primitive-branch11)) compiled-branch10 (assign continue (label after-call9)) (assign val (op compiled-procedure-entry) (reg proc)) (goto (reg val)) primitive-branch11 (assign val (op apply-primitive-procedure) (reg proc) (reg argl)) after-call9 ;; `val` ahora contiene el resultado de `(factorial (- n 1))` (restore argl) ;; restaurar lista de argumentos parcial para (assign argl (op cons) (reg val) (reg argl)) (restore proc) ;; restaurar (restore continue) ;; aplicar `*` y devolver su valor (test (op primitive-procedure?) (reg proc)) (branch (label primitive-branch14)) compiled-branch13 ;; tenga en cuenta que un procedimiento compuesto aquí se llama de forma recursiva de cola (assign val (op compiled-procedure-entry) (reg proc)) (goto (reg val)) primitive-branch14 (assign val (op apply-primitive-procedure) (reg proc) (reg argl)) (goto (reg continue)) after-call12 after-if3 after-lambda1 ;; asignar el procedimiento a la variable `factorial` (perform (op define-variable!) (const factorial) (reg val) (reg env)) (assign val (const ok))

Ejercicio 5.35: ¿Qué expresión se compiló para producir el código que se muestra en la Figura 5.18?

Figura 5.18: ↓ Un ejemplo de salida del compilador. Ver Ejercicio 5.35.

(assign val (op make-compiled-procedure) (label entry16) (reg env)) (goto (label after-lambda15)) entry16 (assign env (op compiled-procedure-env) (reg proc)) (assign env (op extend-environment) (const (x)) (reg argl) (reg env)) (assign proc (op lookup-variable-value) (const +) (reg env)) (save continue) (save proc) (save env) (assign proc (op lookup-variable-value) (const g) (reg env)) (save proc) (assign proc (op lookup-variable-value) (const +) (reg env)) (assign val (const 2)) (assign argl (op list) (reg val)) (assign val (op lookup-variable-value) (const x) (reg env)) (assign argl (op cons) (reg val) (reg argl)) (test (op primitive-procedure?) (reg proc)) (branch (label primitive-branch19)) compiled-branch18 (assign continue (label after-call17)) (assign val (op compiled-procedure-entry) (reg proc)) (goto (reg val)) primitive-branch19 (assign val (op apply-primitive-procedure) (reg proc) (reg argl)) after-call17 (assign argl (op list) (reg val)) (restore proc) (test (op primitive-

procedure?) (reg proc)) (branch (label primitive-branch22)) compiled-branch21 (assign continue (label after-call20)) (assign val (op compiled-procedure-entry) (reg proc)) (goto (reg val)) primitive-branch22 (assign val (op apply-primitive-procedure) (reg proc) (reg argl)) after-call20 (assign argl (op list) (reg val)) (restore env) (assign val (op lookup-variable-value) (const x) (reg env)) (assign argl (op cons) (reg val) (reg argl)) (restore proc) (restore continue) (test (op primitive-procedure?) (reg proc)) (branch (label primitive-branch25)) compiled-branch24 (assign val (op compiled-procedure-entry) (reg proc)) (goto (reg val)) primitive-branch25 (assign val (op apply-primitive-procedure) (reg proc) (reg argl)) (goto (reg continue)) after-call23 after-lambda15 (perform (op define-variable!) (const f) (reg val) (reg env)) (assign val (const ok))

Ejercicio 5.36: ¿Qué orden de evaluación produce nuestro compilador para los operandos de una combinación? ¿Es de izquierda a derecha, de derecha a izquierda o algún otro orden? ¿En qué parte del compilador se determina este orden? Modifique el compilador para que produzca algún otro orden de evaluación. (Vea la discusión sobre el orden de evaluación para el evaluador de control explícito en Section 5.4.1). ¿Cómo afecta el cambio del orden de evaluación de los operandos a la eficiencia del código que construye la lista de argumentos?

Ejercicio 5.37: Una forma de entender el mecanismo `preserving` del compilador para optimizar el uso de la pila es ver qué operaciones adicionales se generarían si no usáramos esta idea. Modifique `preserving` para que siempre genere las operaciones `save` y `restore`. Compile algunas expresiones simples e identifique las operaciones de pila innecesarias que se generan. Compare el código con el generado con el mecanismo `preserving` intacto.

Ejercicio 5.38: Nuestro compilador es inteligente para evitar operaciones de pila innecesarias, pero no es nada inteligente cuando se trata de compilar llamadas a los procedimientos primitivos del lenguaje en términos de las operaciones primitivas suministradas por la máquina. Por ejemplo, considere cuánto código se compila para calcular `(+ a 1)`: El código configura una lista de argumentos en `argl`, pone el procedimiento de suma primitivo (que encuentra buscando el símbolo `+` en el entorno) en `proc`, y prueba si el procedimiento es primitivo o compuesto. El compilador siempre genera código para realizar la prueba, así como código para ramas primitivas y compuestas (solo una de las cuales se ejecutará). No hemos mostrado la parte del controlador que implementa las primitivas, pero presumimos que estas instrucciones hacen uso de operaciones aritméticas primitivas en las rutas de datos de la máquina. Considere cuánto menos código se generaría si el compilador pudiera

codificar primitivas en línea—es decir, si pudiera generar código para usar directamente estas operaciones primitivas de la máquina. La expresión `(+ a 1)` podría ser compilada en algo tan simple como[43]

(assign val (op lookup-variable-value) (const a) (reg env)) (assign val (op +) (reg val) (const 1))

En este ejercicio extenderemos nuestro compilador para admitir la codificación en línea de primitivas seleccionadas. Se generará código de propósito especial para las llamadas a estos procedimientos primitivos en lugar del código general de aplicación de procedimientos. Para soportar esto, aumentaremos nuestra máquina con registros de argumentos especiales `arg1` y `arg2`. Las operaciones aritméticas primitivas de la máquina tomarán sus entradas de `arg1` y `arg2`. Los resultados pueden ponerse en `val`, `arg1` o `arg2`. El compilador debe ser capaz de reconocer la aplicación de una primitiva codificada en línea en el programa fuente. Aumentaremos el despacho en el procedimiento `compile` para reconocer los nombres de estas primitivas además de las palabras reservadas (las formas especiales) que actualmente reconoce.[44] Para cada forma especial, nuestro compilador tiene un generador de código. En este ejercicio construiremos una familia de generadores de código para las primitivas codificadas en línea.

1. Las primitivas codificadas en línea, a diferencia de las formas especiales, todas necesitan que sus operandos sean evaluados. Escriba un generador de código `spread/arguments` para que lo usen todos los generadores de código de codificación en línea. `spread/arguments` debe tomar una lista de operandos y compilar los operandos dados dirigidos a registros de argumentos sucesivos. Tenga en cuenta que un operando puede contener una llamada a una primitiva codificada en línea, por lo que los registros de argumentos tendrán que ser preservados durante la evaluación del operando.

2. Para cada uno de los procedimientos primitivos `=`, `*`, `-` y `+`, escriba un generador de código que tome una combinación con ese operador, junto con un destino y un descriptor de enlace, y produzca código para distribuir los argumentos en los registros y luego realizar la operación dirigida al destino dado con el enlace dado. Solo necesita manejar expresiones con dos operandos. Haga que `compile` despache a estos generadores de código.

3. Pruebe su nuevo compilador con el ejemplo de `factorial`. Compare el código resultante con el resultado producido sin codificación en línea.

4. Extienda sus generadores de código para `+` y `*` para que puedan manejar expresiones con un número arbitrario de operandos. Una expresión con más de dos operandos tendrá que ser compilada en una secuencia de operaciones, cada una con solo dos entradas.

Direccionamiento Léxico

Una de las optimizaciones más comunes realizadas por los compiladores es la optimización de la búsqueda de variables. Nuestro compilador, tal como lo hemos implementado hasta ahora, genera código que utiliza la operación `lookup/variable/value` de la máquina evaluadora. Esto busca una variable comparándola con cada variable que está actualmente vinculada, trabajando marco por marco hacia afuera a través del entorno de tiempo de ejecución. Esta búsqueda puede ser costosa si los marcos están profundamente anidados o si hay muchas variables. Por ejemplo, considere el problema de buscar el valor de `x` mientras se evalúa la expresión `(* x y z)` en una aplicación del procedimiento que es devuelto por

(let ((x 3) (y 4)) (lambda (a b c d e) (let ((y (* a b x)) (z (+ c d x))) (* x y z))))

Como una expresión `let` es solo azúcar sintáctico para una combinación `lambda`, esta expresión es equivalente a

((lambda (x y) (lambda (a b c d e) ((lambda (y z) (* x y z)) (* a b x) (+ c d x)))) 3 4)

Cada vez que `lookup/variable/value` busca `x`, debe determinar que el símbolo `x` no es `eq?` a `y` o `z` (en el primer marco), ni a `a`, `b`, `c`, `d`, o `e` (en el segundo marco). Asumiremos, por el momento, que nuestros programas no usan `define`—que las variables solo se vinculan con `lambda`. Debido a que nuestro lenguaje tiene alcance léxico, el entorno de tiempo de ejecución para cualquier expresión tendrá una estructura que es paralela a la estructura léxica del programa en el que aparece la expresión.[45] Por lo tanto, el compilador puede saber, cuando analiza la expresión anterior, que cada vez que se aplique el procedimiento, la variable `x` en `(* x y z)` se encontrará dos marcos hacia afuera desde el marco actual y será la primera variable en ese marco. Podemos aprovechar este hecho inventando un nuevo tipo de operación de búsqueda de variables, `lexical/address/lookup`, que toma como argumentos un entorno y una *dirección léxica* que consta de dos números: un *número de marco*, que especifica cuántos marcos pasar por alto, y un *número de desplazamiento*, que especifica cuántas variables pasar por alto en ese marco. `lexical/address/lookup` producirá el valor de la variable almacenada en esa dirección léxica relativa al entorno actual. Si agregamos

la operación `lexical/address/lookup` a nuestra máquina, podemos hacer que el compilador genere código que haga referencia a las variables usando esta operación, en lugar de `lookup/variable/value`. De manera similar, nuestro código compilado puede usar una nueva operación `lexical/address/set!` en lugar de `set/variable/value!`. Para generar dicho código, el compilador debe ser capaz de determinar la dirección léxica de una variable a la que está a punto de compilar una referencia. La dirección léxica de una variable en un programa depende de dónde se encuentre uno en el código. Por ejemplo, en el siguiente programa, la dirección de `x` en la expresión ⟨*e1*$\kern0.08em\rangle$ es (2, 0)—dos marcos atrás y la primera variable en el marco. En ese punto, `y` está en la dirección (0, 0) y `c` está en la dirección (1, 2). En la expresión ⟨*e2*$\kern0.09em\rangle$, `x` está en (1, 0), `y` está en (1, 1), y `c` está en (0, 2).

((lambda (x y) (lambda (a b c d e) ((lambda (y z) ⟨ *e1* ⟩) ⟨ *e2* ⟩ (+ c d x)))) 3 4)

Una forma para que el compilador produzca código que use direccionamiento léxico es mantener una estructura de datos llamada *entorno de tiempo de compilación*. Esto mantiene un registro de qué variables estarán en qué posiciones en qué marcos en el entorno de tiempo de ejecución cuando se ejecute una operación de acceso a variable particular. El entorno de tiempo de compilación es una lista de marcos, cada uno conteniendo una lista de variables. (Por supuesto, no habrá valores vinculados a las variables, ya que los valores no se calculan en tiempo de compilación). El entorno de tiempo de compilación se convierte en un argumento adicional para `compile` y se pasa a cada generador de código. La llamada de nivel superior a `compile` utiliza un entorno de tiempo de compilación vacío. Cuando se compila un cuerpo de `lambda`, `compile/lambda/body` extiende el entorno de tiempo de compilación con un marco que contiene los parámetros del procedimiento, de modo que la secuencia que compone el cuerpo se compila con ese entorno extendido. En cada punto de la compilación, `compile/variable` y `compile/assignment` usan el entorno de tiempo de compilación para generar las direcciones léxicas apropiadas. Del Ejercicio 5.39 al Ejercicio 5.43 se describe cómo completar este bosquejo de la estrategia de direccionamiento léxico para incorporar la búsqueda léxica en el compilador. El Ejercicio 5.44 describe otro uso para el entorno de tiempo de compilación.

> **Ejercicio 5.39:** Escriba un procedimiento `lexical/address/lookup` que implemente la nueva operación de búsqueda. Debe tomar dos argumentos —una dirección léxica y un entorno de tiempo de ejecución—y devolver el valor de la variable almacenada en la dirección léxica especificada. `lexical/address/lookup` debe señalar un error si el valor de la variable es el símbolo `unassigned`.[46] También escriba un procedimiento

`lexical/address/set!` que implemente la operación que cambia el valor de la variable en una dirección léxica especificada.

Ejercicio 5.40: Modifique el compilador para mantener el entorno de tiempo de compilación como se describe anteriormente. Es decir, agregue un argumento de entorno de tiempo de compilación a `compile` y los diversos generadores de código, y extiéndalo en `compile/lambda/body`.

Ejercicio 5.41: Escriba un procedimiento `find/variable` que tome como argumentos una variable y un entorno de tiempo de compilación y devuelva la dirección léxica de la variable con respecto a ese entorno. Por ejemplo, en el fragmento de programa que se muestra arriba, el entorno de tiempo de compilación durante la compilación de la expresión ⟨*e1*$ \kern0.08em\rangle$ es `((y z) (a b c d e) (x y))`. `find/variable` debería producir

(find-variable 'c '((y z) (a b c d e) (x y))) *(1 2)* (find-variable 'x '((y z) (a b c d e) (x y))) *(2 0)* (find-variable 'w '((y z) (a b c d e) (x y))) *no-encontrado*

Ejercicio 5.42: Usando `find/variable` del Ejercicio 5.41, reescriba `compile/variable` y `compile/assignment` para que generen instrucciones de dirección léxica. En los casos en que `find/variable` devuelva `not/found` (es decir, donde la variable no está en el entorno de tiempo de compilación), debe hacer que los generadores de código usen las operaciones del evaluador, como antes, para buscar la vinculación. (El único lugar donde una variable que no se encuentra en tiempo de compilación puede estar es en el entorno global, que es parte del entorno de tiempo de ejecución pero no es parte del entorno de tiempo de compilación.[47] Por lo tanto, si lo desea, puede hacer que las operaciones del evaluador busquen directamente en el entorno global, que se puede obtener con la operación `(op get/global/environment)`, en lugar de hacer que busquen en todo el entorno de tiempo de ejecución que se encuentra en `env`.) Pruebe el compilador modificado en algunos casos simples, como la combinación anidada `lambda` al principio de esta sección.

Ejercicio 5.43: Argumentamos en Section 4.1.6 que las definiciones internas para la estructura de bloques no deben considerarse `define`s "reales". Más bien, el cuerpo de un procedimiento debe interpretarse como si las variables internas que se definen se instalaran como variables `lambda` ordinarias inicializadas a sus valores correctos usando `set!`. El Section 4.1.6 y el Ejercicio 4.16 mostraron cómo modificar el intérprete metacircular para lograr esto escaneando las definiciones internas.

Modifique el compilador para realizar la misma transformación antes de compilar el cuerpo de un procedimiento.

Ejercicio 5.44: En esta sección nos hemos centrado en el uso del entorno de tiempo de compilación para producir direcciones léxicas. Pero hay otros usos para los entornos de tiempo de compilación. Por ejemplo, en el Ejercicio 5.38 aumentamos la eficiencia del código compilado codificando en línea los procedimientos primitivos. Nuestra implementación trató los nombres de los procedimientos codificados en línea como palabras reservadas. Si un programa redefiniera tal nombre, el mecanismo descrito en el Ejercicio 5.38 seguiría codificándolo en línea como una primitiva, ignorando la nueva vinculación. Por ejemplo, considere el procedimiento

(lambda (+ * a b x y) (+ (* a x) (* b y)))

que calcula una combinación lineal de `x` e `y`. Podríamos llamarlo con los argumentos `+matrix`, `matrix`, y cuatro matrices, pero el compilador de codificación en línea seguiría codificando en línea el `+` y el en `(+ (* a x) (* b y))` como `+` y `*` primitivos. Modifique el compilador de codificación en línea para que consulte el entorno de tiempo de compilación para compilar el código correcto para las expresiones que involucran los nombres de los procedimientos primitivos. (El código funcionará correctamente siempre que el programa no haga `define` o `set!` a estos nombres).

Interfaz del Código Compilado con el Evaluador

Aún no hemos explicado cómo cargar el código compilado en la máquina evaluadora ni cómo ejecutarlo. Asumiremos que la máquina evaluadora de control explícito ha sido definida como en Section 5.4.4, con las operaciones adicionales especificadas en la Nota a pie de página 38. Implementaremos un procedimiento `compile/and/go` que compila una expresión de Scheme, carga el código objeto resultante en la máquina evaluadora, y hace que la máquina ejecute el código en el entorno global del evaluador, imprima el resultado, y entre en el bucle principal del evaluador. También modificaremos el evaluador para que las expresiones interpretadas puedan llamar a procedimientos compilados así como a los interpretados. Luego podemos poner un procedimiento compilado en la máquina y usar el evaluador para llamarlo:

(compile-and-go '(define (factorial n) (if (= n 1) 1 (* (factorial (- n 1)) n))))
;;; Valor EC-Eval: ok ;;; Entrada EC-Eval: (factorial 5) *;;; Valor EC-Eval:*

Para permitir que el evaluador maneje procedimientos compilados (por ejemplo, para evaluar la llamada a `factorial` anterior), necesitamos cambiar el código en `apply/dispatch` (Section 5.4.1) para que reconozca procedimientos compilados (distintos de los procedimientos compuestos o primitivos) y transfiera el control directamente al punto de entrada del código compilado:[48]

apply-dispatch (test (op primitive-procedure?) (reg proc)) (branch (label primitive-apply)) (test (op compound-procedure?) (reg proc)) (branch (label compound-apply)) (test (op compiled-procedure?) (reg proc)) (branch (label compiled-apply)) (goto (label unknown-procedure-type)) compiled-apply (restore continue) (assign val (op compiled-procedure-entry) (reg proc)) (goto (reg val))

Note la restauración de `continue` en `compiled/apply`. Recuerde que el evaluador se organizó de modo que en `apply/dispatch`, la continuación estaría en la parte superior de la pila. El punto de entrada del código compilado, por otro lado, espera que la continuación esté en `continue`, por lo que `continue` debe ser restaurado antes de que se ejecute el código compilado. Para permitirnos ejecutar algún código compilado cuando iniciamos la máquina evaluadora, agregamos una instrucción `branch` al principio de la máquina evaluadora, que hace que la máquina vaya a un nuevo punto de entrada si el registro `flag` está establecido.[49]

(branch (label external-entry)) ; salta si `flag` está establecido read-eval-print-loop (perform (op initialize-stack)) …

`external/entry` asume que la máquina se inicia con `val` conteniendo la ubicación de una secuencia de instrucciones que pone un resultado en `val` y termina con `(goto (reg continue))`. Iniciar en este punto de entrada salta a la ubicación designada por `val`, pero primero asigna `continue` para que la ejecución regrese a `print/result`, que imprime el valor en `val` y luego va al principio del bucle de lectura-evaluación-impresión del evaluador.[50]

external-entry (perform (op initialize-stack)) (assign env (op get-global-environment)) (assign continue (label print-result)) (goto (reg val))

Ahora podemos usar el siguiente procedimiento para compilar la definición de un procedimiento, ejecutar el código compilado, y ejecutar el bucle de lectura-evaluación-impresión para que podamos probar el procedimiento. Como queremos que el código compilado regrese a la ubicación en `continue` con su resultado en `val`, compilamos la expresión con un destino de `val` y un enlace de `return`. Para transformar el código objeto producido por el compilador en instrucciones ejecutables para la máquina de registros del

evaluador, usamos el procedimiento `assemble` del simulador de máquina de registros (Section 5.2.2). Luego inicializamos el registro `val` para que apunte a la lista de instrucciones, establecemos el `flag` para que el evaluador vaya a `external/entry`, e iniciamos el evaluador.

(define (compile-and-go expression) (let ((instructions (assemble (statements (compile expression 'val 'return)) eceval))) (set! the-global-environment (setup-environment)) (set-register-contents! eceval 'val instructions) (set-register-contents! eceval 'flag true) (start eceval)))

Si hemos configurado el monitoreo de la pila, como al final de Section 5.4.4, podemos examinar el uso de la pila del código compilado:

(compile-and-go '(define (factorial n) (if (= n 1) 1 (* (factorial (- n 1)) n)))) *(total-pushes = 0 maximum-depth = 0) ;;; Valor EC-Eval: ok ;;; Entrada EC-Eval:* (factorial 5) *(total-pushes = 31 maximum-depth = 14) ;;; Valor EC-Eval: 120*

Compare este ejemplo con la evaluación de `(factorial 5)` usando la versión interpretada del mismo procedimiento, mostrada al final de Section 5.4.4. La versión interpretada requirió 144 pushes y una profundidad máxima de pila de 28. Esto ilustra la optimización que resulta de nuestra estrategia de compilación.

Interpretación y compilación

Con los programas de esta sección, ahora podemos experimentar con las estrategias de ejecución alternativas de interpretación y compilación.[51] Un intérprete eleva la máquina al nivel del programa del usuario; un compilador baja el programa del usuario al nivel del lenguaje de la máquina. Podemos considerar el lenguaje Scheme (o cualquier lenguaje de programación) como una familia coherente de abstracciones erigidas sobre el lenguaje de la máquina. Los intérpretes son buenos para el desarrollo interactivo de programas y la depuración porque los pasos de la ejecución del programa se organizan en términos de estas abstracciones, y por lo tanto son más inteligibles para el programador. El código compilado puede ejecutarse más rápido, porque los pasos de la ejecución del programa se organizan en términos del lenguaje de la máquina, y el compilador es libre de hacer optimizaciones que atraviesan las abstracciones de nivel superior.[52] Las alternativas de interpretación y compilación también conducen a diferentes estrategias para portar lenguajes a nuevas computadoras. Supongamos que deseamos implementar Lisp para una nueva máquina. Una estrategia es comenzar con el evaluador de control explícito de Section 5.4 y traducir sus instrucciones a instrucciones para la nueva máquina. Una estrategia diferente

es comenzar con el compilador y cambiar los generadores de código para que generen código para la nueva máquina. La segunda estrategia nos permite ejecutar cualquier programa Lisp en la nueva máquina compilándolo primero con el compilador ejecutándose en nuestro sistema Lisp original, y enlazándolo con una versión compilada de la biblioteca de tiempo de ejecución. [53] Mejor aún, podemos compilar el propio compilador, y ejecutarlo en la nueva máquina para compilar otros programas de Lisp.[54] O podemos compilar uno de los intérpretes de Section 4.1 para producir un intérprete que se ejecute en la nueva máquina.

Ejercicio 5.45: Comparando las operaciones de pila usadas por el código compilado con las operaciones de pila usadas por el evaluador para la misma computación, podemos determinar en qué medida el compilador optimiza el uso de la pila, tanto en velocidad (reduciendo el número total de operaciones de pila) como en espacio (reduciendo la profundidad máxima de la pila). Comparar este uso optimizado de la pila con el rendimiento de una máquina de propósito especial para la misma computación da una indicación de la calidad del compilador.

1. El Ejercicio 5.27 le pidió que determinara, como función de *n*, el número de pushes y la profundidad máxima de la pila necesarios por el evaluador para calcular *n*! usando el procedimiento factorial recursivo dado anteriormente. El Ejercicio 5.14 le pidió que hiciera las mismas mediciones para la máquina factorial de propósito especial que se muestra en la Figura 5.11. Ahora realice el mismo análisis usando el procedimiento `factorial` compilado. Tome la razón del número de pushes en la versión compilada al número de pushes en la versión interpretada, y haga lo mismo para la profundidad máxima de la pila. Dado que el número de operaciones y la profundidad de la pila utilizada para calcular *n*! son lineales en *n*, estas razones deberían acercarse a constantes a medida que *n* se vuelve grande. ¿Cuáles son estas constantes? De manera similar, encuentre las razones del uso de la pila en la máquina de propósito especial al uso en la versión interpretada. Compare las razones para el código de propósito especial versus el interpretado con las razones para el código compilado versus el interpretado. Debería encontrar que la máquina de propósito especial funciona mucho mejor que el código compilado, ya que el código del controlador hecho a mano debería ser mucho mejor que lo que produce nuestro rudimentario compilador de propósito general.

2. ¿Puede sugerir mejoras al compilador que le ayudarían a generar código que se acercaría más en rendimiento a la versión hecha a mano?

Ejercicio 5.46: Realice un análisis como el del Ejercicio 5.45 para determinar la efectividad de compilar el procedimiento de Fibonacci recursivo de árbol

(define (fib n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))

comparado con la efectividad de usar la máquina de Fibonacci de propósito especial de la Figura 5.12. (Para la medición del rendimiento interpretado, vea el Ejercicio 5.29). Para Fibonacci, el recurso de tiempo utilizado no es lineal en *n*; por lo tanto, las razones de las operaciones de pila no se acercarán a un valor límite que sea independiente de *n*.

Ejercicio 5.47: Esta sección describió cómo modificar el evaluador de control explícito para que el código interpretado pueda llamar a procedimientos compilados. Muestre cómo modificar el compilador para que los procedimientos compilados puedan llamar no solo a procedimientos primitivos y procedimientos compilados, sino también a procedimientos interpretados. Esto requiere modificar `compile/procedure/call` para manejar el caso de procedimientos compuestos (interpretados). Asegúrese de manejar todas las mismas combinaciones de `target` y `linkage` que en `compile/proc/appl`. Para hacer la aplicación real del procedimiento, el código necesita saltar al punto de entrada `compound/apply` del evaluador. Esta etiqueta no puede ser referenciada directamente en el código objeto (ya que el ensamblador requiere que todas las etiquetas referenciadas por el código que está ensamblando estén definidas allí), por lo que agregaremos un registro llamado `compapp` a la máquina evaluadora para mantener este punto de entrada, y agregaremos una instrucción para inicializarlo:

(assign compapp (label compound-apply)) (branch (label external-entry)) ; salta si `flag` está establecido read-eval-print-loop …

Para probar su código, comience definiendo un procedimiento `f` que llama a un procedimiento `g`. Use `compile/and/go` para compilar la definición de `f` e iniciar el evaluador. Ahora, escribiendo en el evaluador, defina `g` e intente llamar a `f`.

Ejercicio 5.48: La interfaz `compile/and/go` implementada en esta sección es incómoda, ya que el compilador solo puede ser llamado una vez (cuando se inicia la máquina evaluadora). Aumente la interfaz compilador-intérprete proporcionando una primitiva `compile/and/run` que pueda ser llamada desde dentro del evaluador de control explícito de la siguiente manera:

;;; Entrada EC-Eval: (compile-and-run '(define (factorial n) (if (= n 1) 1

(* (factorial (- n 1)) n)))) *;;; Valor EC-Eval: ok ;;; Entrada EC-Eval:* (factorial 5) *;;; Valor EC-Eval: 120*

Ejercicio 5.49: Como alternativa a usar el bucle de lectura-evaluación-impresión del evaluador de control explícito, diseñe una máquina de registros que realice un bucle de lectura-compilación-ejecución-impresión. Es decir, la máquina debe ejecutar un bucle que lee una expresión, la compila, ensambla y ejecuta el código resultante, e imprime el resultado. Esto es fácil de ejecutar en nuestra configuración simulada, ya que podemos hacer arreglos para llamar a los procedimientos `compile` y `assemble` como "operaciones de máquina de registros".

Ejercicio 5.50: Use el compilador para compilar el evaluador metacircular de Section 4.1 y ejecute este programa usando el simulador de máquina de registros. (Para compilar más de una definición a la vez, puede empaquetar las definiciones en un `begin`). El intérprete resultante se ejecutará muy lentamente debido a los múltiples niveles de interpretación, pero lograr que todos los detalles funcionen es un ejercicio instructivo.

Ejercicio 5.51: Desarrolle una implementación rudimentaria de Scheme en C (o algún otro lenguaje de bajo nivel de su elección) traduciendo el evaluador de control explícito de Section 5.4 a C. Para ejecutar este código, también necesitará proporcionar rutinas de asignación de almacenamiento apropiadas y otro soporte en tiempo de ejecución.

Ejercicio 5.52: Como contrapunto al Ejercicio 5.51, modifique el compilador para que compile procedimientos de Scheme en secuencias de instrucciones C. Compile el evaluador metacircular de Section 4.1 para producir un intérprete de Scheme escrito en C.

1. Esta suposición pasa por alto una gran cantidad de complejidad. Generalmente, una gran parte de la implementación de un sistema Lisp se dedica a hacer que la lectura y la impresión funcionen.

2. Se podría argumentar que no necesitamos guardar el `n` antiguo; después de decrementarlo y resolver el subproblema, simplemente podríamos incrementarlo para recuperar el valor antiguo. Aunque esta estrategia funciona para el factorial, no puede funcionar en general, ya que el valor antiguo de un registro no siempre se puede calcular a partir del nuevo.

3. En Section 5.3 veremos cómo implementar una pila en términos de operaciones más primitivas.

4. Usar el procedimiento `receive` aquí es una forma de hacer que `extract/labels` devuelva efectivamente dos valores—`labels` e `insts`—sin crear explícitamente una estructura de datos compuesta para contenerlos. Una implementación alternativa, que devuelve un par explícito de valores, es

(define (extract-labels text) (if (null? text) (cons '() '()) (let ((result (extract-labels (cdr text)))) (let ((insts (car result)) (labels (cdr result))) (let ((next-inst (car text))) (if (symbol? next-inst) (cons insts (cons (make-label-entry next-inst insts) labels)) (cons (cons (make-instruction next-inst) insts) labels)))))))

que sería llamado por `assemble` de la siguiente manera:

(define (assemble controller-text machine) (let ((result (extract-labels controller-text))) (let ((insts (car result)) (labels (cdr result))) (update-insts! insts labels machine) insts)))

Puede considerar nuestro uso de `receive` como una demostración de una forma elegante de devolver múltiples valores, or simplemente una excusa para mostrar un truco de programación. Un argumento como `receive` que es el siguiente procedimiento a ser invocado se llama una "continuación". Recuerde que también usamos continuaciones para implementar la estructura de control de retroceso (backtracking) en el evaluador `amb` en Section 4.3.3.

5. Podríamos representar la memoria como listas de elementos. Sin embargo, el tiempo de acceso no sería entonces independiente del índice, ya que acceder al n^{th} elemento de una lista requiere $n - 1$ operaciones `cdr`.

6. Para ser completos, deberíamos especificar una operación `make/vector` que construya vectores. Sin embargo, en la presente aplicación usaremos vectores solo para modelar divisiones fijas de la memoria de la computadora.

7. Esta es precisamente la misma idea de "datos etiquetados" que introdujimos en el Capítulo 2 para tratar con operaciones genéricas. Aquí, sin embargo, los tipos de datos se incluyen en el nivel primitivo de la máquina en lugar de construirse mediante el uso de listas.

8. La información de tipo puede codificarse de varias maneras, dependiendo de los detalles de la máquina en la que se implementará el sistema Lisp. La eficiencia de ejecución de los programas Lisp dependerá en gran medida de cuán inteligentemente se haga esta elección, pero es difícil formular reglas de diseño generales para buenas elecciones. La forma más directa de implementar punteros tipados es asignar un conjunto fijo de bits en cada puntero para que sea un *campo de tipo* que codifique el tipo de dato. Las preguntas importantes a abordar al diseñar tal representación incluyen las siguientes: ¿Cuántos bits de tipo se requieren? ¿Qué tan grandes deben ser

los índices de vector? ¿Con qué eficiencia se pueden usar las instrucciones primitivas de la máquina para manipular los campos de tipo de los punteros? Las máquinas que incluyen hardware especial para el manejo eficiente de los campos de tipo se dice que tienen *arquitecturas etiquetadas*.

9. Esta decisión sobre la representación de los números determina si `eq?`, que prueba la igualdad de punteros, puede usarse para probar la igualdad de números. Si el puntero contiene el número mismo, entonces los números iguales tendrán el mismo puntero. Pero si el puntero contiene el índice de una ubicación donde se almacena el número, solo se garantizará que los números iguales tengan punteros iguales si tenemos cuidado de nunca almacenar el mismo número en más de una ubicación.

10. Esto es como escribir un número como una secuencia de dígitos, excepto que cada "dígito" es un número entre 0 y el número más grande que se puede almacenar en un solo puntero.

11. Hay otras formas de encontrar almacenamiento libre. Por ejemplo, podríamos enlazar todos los pares no utilizados en una *lista libre*. Nuestras ubicaciones libres son consecutivas (y por lo tanto se puede acceder a ellas incrementando un puntero) porque estamos usando un recolector de basura compactador, como veremos en Section 5.3.2.

12. Esto es esencialmente la implementación de `cons` en términos de `set/car!` y `set/cdr!`, como se describe en Section 3.3.1. La operación `get/new/pair` utilizada en esa implementación se realiza aquí por el puntero `free`.

13. Esto puede no ser cierto eventualmente, porque las memorias pueden llegar a ser tan grandes que sería imposible quedarse sin memoria libre en la vida útil de la computadora. Por ejemplo, hay alrededor de $3 \cdot 10^{13}$ microsegundos en un año, por lo que si hiciéramos un `cons` una vez por microsegundo, necesitaríamos alrededor de 10^{15} celdas de memoria para construir una máquina que pudiera operar durante 30 años sin quedarse sin memoria. Esa cantidad de memoria parece absurdamente grande para los estándares actuales, pero no es físicamente imposible. Por otro lado, los procesadores se están volviendo más rápidos y una futura computadora podría tener un gran número de procesadores operando en paralelo en una sola memoria, por lo que podría ser posible agotar la memoria mucho más rápido de lo que hemos postulado.

14. Asumimos aquí que la pila se representa como una lista como se describe en Section 5.3.1, de modo que los elementos en la pila son accesibles a través del puntero en el registro de la pila.

15. Esta idea fue inventada e implementada por primera vez por Minsky,

como parte de la implementación de Lisp para el PDP-1 en el Laboratorio de Investigación de Electrónica del MIT. Fue desarrollada posteriormente por Fenichel y Yochelson (1969) para su uso en la implementación de Lisp para el sistema de tiempo compartido Multics. Más tarde, Baker (1978) desarrolló una versión de "tiempo real" del método, que no requiere que el cálculo se detenga durante la recolección de basura. La idea de Baker fue extendida por Hewitt, Lieberman y Moon (ver Lieberman y Hewitt 1983) para aprovechar el hecho de que algunas estructuras son más volátiles y otras más permanentes.

Una técnica de recolección de basura comúnmente utilizada alternativa es el método *mark-sweep*. Este consiste en rastrear toda la estructura accesible desde los registros de la máquina y marcar cada par que alcanzamos. Luego escaneamos toda la memoria, y cualquier ubicación que no esté marcada es "barrida" como basura y puesta a disposición para su reutilización. Se puede encontrar una discusión completa del método mark-sweep en Allen 1978.

El algoritmo Minsky-Fenichel-Yochelson es el algoritmo dominante en uso para sistemas de memoria grande porque examina solo la parte útil de la memoria. Esto contrasta con mark-sweep, en el que la fase de barrido debe verificar toda la memoria. Una segunda ventaja de stop-and-copy es que es un recolector de basura *compactador*. Es decir, al final de la fase de recolección de basura, los datos útiles se habrán movido a ubicaciones de memoria consecutivas, con todos los pares de basura comprimidos. Esto puede ser una consideración de rendimiento extremadamente importante en máquinas con memoria virtual, en las que los accesos a direcciones de memoria muy separadas pueden requerir operaciones de paginación adicionales.

16. Esta lista de registros no incluye los registros utilizados por el sistema de asignación de almacenamiento—`root`, `the/cars`, `the/cdrs`, y los otros registros que se introducirán en esta sección.

17. El término *corazón roto* fue acuñado por David Cressey, quien escribió un recolector de basura para MDL, un dialecto de Lisp desarrollado en el MIT durante la década de 1970.

18. El recolector de basura utiliza el predicado de bajo nivel `pointer/to/pair?` en lugar de la operación de estructura de lista `pair?` porque en un sistema real podría haber varias cosas que se tratan como pares para fines de recolección de basura. Por ejemplo, en un sistema Scheme que se ajusta al estándar IEEE, un objeto de procedimiento puede implementarse como un tipo especial de "par" que no satisface el predicado `pair?`. Para fines de simulación, `pointer/to/pair?` puede implementarse como `pair?`.

19. Ver Batali et al. 1982 para más información sobre el chip y el método por el cual fue diseñado.

20. En nuestro controlador, el despacho se escribe como una secuencia de instrucciones `test` y `branch`. Alternativamente, podría haberse escrito en un estilo dirigido por datos (y en un sistema real probablemente lo habría sido) para evitar la necesidad de realizar pruebas secuenciales y para facilitar la definición de nuevos tipos de expresiones. Una máquina diseñada para ejecutar Lisp probablemente incluiría una instrucción `dispatch/on/type` que ejecutaría eficientemente tales despachos dirigidos por datos.

21. Este es un punto importante pero sutil al traducir algoritmos de un lenguaje procedural, como Lisp, a un lenguaje de máquina de registros. Como alternativa a guardar solo lo que se necesita, podríamos guardar todos los registros (excepto `val`) antes de cada llamada recursiva. Esto se llama una disciplina de *pila enmarcada*. Esto funcionaría pero podría guardar más registros de los necesarios; esto podría ser una consideración importante en un sistema donde las operaciones de pila son costosas. Guardar registros cuyo contenido no se necesitará más tarde también puede retener datos inútiles que de otro modo podrían ser recolectados como basura, liberando espacio para ser reutilizado.

22. Agregamos a los procedimientos de estructura de datos del evaluador en Section 4.1.3 los siguientes dos procedimientos para manipular listas de argumentos:

(define (empty-arglist) '()) (define (adjoin-arg arg arglist) (append arglist (list arg)))

También usamos un procedimiento de sintaxis adicional para probar el último operando en una combinación:

(define (last-operand? ops) (null? (cdr ops)))

23. La optimización de tratar el último operando de forma especial se conoce como *recursión de cola de evlis* (ver Wand 1980). Podríamos ser algo más eficientes en el bucle de evaluación de argumentos si hiciéramos de la evaluación del primer operando un caso especial también. Esto nos permitiría posponer la inicialización de `argl` hasta después de evaluar el primer operando, para evitar guardar `argl` en este caso. El compilador en Section 5.5 realiza esta optimización. (Compare el procedimiento `construct/arglist` de Section 5.5.3.)

24. El orden de evaluación de operandos en el evaluador metacircular está determinado por el orden de evaluación de los argumentos de `cons` en el

procedimiento `list/of/values` de Section 4.1.1 (ver Ejercicio 4.1).

25. Vimos en Section 5.1 cómo implementar tal proceso con una máquina de registros que no tenía pila; el estado del proceso se almacenaba en un conjunto fijo de registros.

26. Esta implementación de la recursión de cola en `ev/sequence` es una variedad de una técnica de optimización bien conocida utilizada por muchos compiladores. Al compilar un procedimiento que termina con una llamada a procedimiento, se puede reemplazar la llamada por un salto al punto de entrada del procedimiento llamado. Incorporar esta estrategia en el intérprete, como hemos hecho en esta sección, proporciona la optimización de manera uniforme en todo el lenguaje.

27. Podemos definir `no/more/exps?` de la siguiente manera:

(define (no-more-exps? seq) (null? seq))

28. Esto no es realmente hacer trampa. En una implementación real desde cero, usaríamos nuestro evaluador de control explícito para interpretar un programa de Scheme que realiza transformaciones a nivel de fuente como `cond/>if` en una fase de sintaxis que se ejecuta antes de la ejecución.

29. Asumimos aquí que `read` y las diversas operaciones de impresión están disponibles como operaciones primitivas de la máquina, lo cual es útil para nuestra simulación, pero completamente irrealista en la práctica. Estas son en realidad operaciones extremadamente complejas. En la práctica, se implementarían usando operaciones de entrada/salida de bajo nivel como transferir caracteres individuales hacia y desde un dispositivo.

Para soportar la operación `get/global/environment` definimos

(define the-global-environment (setup-environment)) (define (get-global-environment) the-global-environment)

30. Hay otros errores que nos gustaría que el intérprete manejara, pero estos no son tan simples. Ver Ejercicio 5.30.

31. Podríamos realizar la inicialización de la pila solo después de los errores, pero hacerlo en el bucle del conductor será conveniente para monitorear el rendimiento del evaluador, como se describe a continuación.

32. Lamentablemente, este es el estado normal de las cosas en sistemas de lenguaje basados en compiladores convencionales como C. En UNIX(tm) el sistema "vuelca el núcleo" (dumps core), y en DOS/Windows(tm) se vuelve

catatónico. El Macintosh(tm) muestra una imagen de una bomba explotando y le ofrece la oportunidad de reiniciar la computadora—si tiene suerte.

33. Esta es una declaración teórica. No estamos afirmando que las rutas de datos del evaluador sean un conjunto particularmente conveniente o eficiente de rutas de datos para una computadora de propósito general. Por ejemplo, no son muy buenas para implementar cálculos de punto flotante de alto rendimiento o cálculos que manipulan intensivamente vectores de bits.

34. En realidad, la máquina que ejecuta el código compilado puede ser más simple que la máquina intérprete, porque no usaremos los registros `exp` y `unev`. El intérprete los usaba para mantener partes de expresiones no evaluadas. Con el compilador, sin embargo, estas expresiones se incorporan en el código compilado que la máquina de registros ejecutará. Por la misma razón, no necesitamos las operaciones de la máquina que tratan con la sintaxis de las expresiones. Pero el código compilado usará algunas operaciones adicionales de la máquina (para representar objetos de procedimiento compilados) que no aparecieron en la máquina del evaluador de control explícito.

35. Observe, sin embargo, que nuestro compilador es un programa de Scheme, y los procedimientos de sintaxis que utiliza para manipular expresiones son los procedimientos reales de Scheme utilizados con el evaluador metacircular. Para el evaluador de control explícito, en contraste, asumimos que operaciones de sintaxis equivalentes estaban disponibles como operaciones para la máquina de registros. (Por supuesto, cuando simulamos la máquina de registros en Scheme, usamos los procedimientos reales de Scheme en nuestra simulación de la máquina de registros).

36. Este procedimiento utiliza una característica de Lisp llamada *comilla inversa* (o *cuasi-cita*) que es útil para construir listas. Preceder una lista con un símbolo de comilla inversa es muy parecido a citarla, excepto que cualquier cosa en la lista que esté marcada con una coma se evalúa. Por ejemplo, si el valor de `linkage` es el símbolo `branch25`, entonces la expresión

‘((goto (label ,linkage)))

se evalúa a la lista

((goto (label branch25)))

De manera similar, si el valor de `x` es la lista `(a b c)`, entonces

‘(1 2 ,(car x))

se evalúa a la lista

(1 2 a).

37. No podemos simplemente usar las etiquetas `true/branch`, `false/branch` y `after/if` como se muestra arriba, porque podría haber más de un `if` en el programa. El compilador usa el procedimiento `make/label` para generar etiquetas. `make/label` toma un símbolo como argumento y devuelve un nuevo símbolo que comienza con el símbolo dado. Por ejemplo, llamadas sucesivas a `(make/label 'a)` devolverían `a1`, `a2`, y así sucesivamente. `make/label` se puede implementar de manera similar a la generación de nombres de variables únicos en el lenguaje de consulta, de la siguiente manera:

(define label-counter 0) (define (new-label-number) (set! label-counter (+ 1 label-counter)) label-counter) (define (make-label name) (string->symbol (string-append (symbol->string name) (number->string (new-label-number)))))

38. Necesitamos operaciones de máquina para implementar una estructura de datos para representar procedimientos compilados, análoga a la estructura para procedimientos compuestos descrita en Section 4.1.3:

(define (make-compiled-procedure entry env) (list 'compiled-procedure entry env)) (define (compiled-procedure? proc) (tagged-list? proc 'compiled-procedure)) (define (compiled-procedure-entry c-proc) (cadr c-proc)) (define (compiled-procedure-env c-proc) (caddr c-proc))

39. En realidad, señalamos un error cuando el destino no es `val` y el enlace es `return`, ya que el único lugar donde solicitamos enlaces `return` es al compilar procedimientos, y nuestra convención es que los procedimientos devuelven sus valores en `val`.

40. Hacer que un compilador genere código recursivo de cola puede parecer una idea sencilla. Pero la mayoría de los compiladores para lenguajes comunes, incluyendo C y Pascal, no hacen esto, y por lo tanto estos lenguajes no pueden representar procesos iterativos solo en términos de llamada a procedimiento. La dificultad con la recursión de cola en estos lenguajes es que sus implementaciones usan la pila para almacenar argumentos de procedimiento y variables locales, así como direcciones de retorno. Las implementaciones de Scheme descritas en este libro almacenan argumentos y variables en memoria para ser recolectados como basura. La razón para usar la pila para variables y argumentos es que evita la necesidad de recolección de basura en lenguajes que de otro modo no la requerirían, y generalmente se cree que es más eficiente. Los compiladores de Lisp sofisticados pueden, de hecho, usar la pila para argumentos sin destruir la recursión de cola. (Ver

Hanson 1990 para una descripción). También hay cierto debate sobre si la asignación en pila es realmente más eficiente que la recolección de basura en primer lugar, pero los detalles parecen depender de puntos finos de la arquitectura de la computadora. (Ver Appel 1987 y Miller and Rozas 1994 para puntos de vista opuestos sobre este tema).

41. La variable `all/regs` está vinculada a la lista de nombres de todos los registros:

(define all-regs '(env proc val argl continue))

42. Tenga en cuenta que `preserving` llama a `append` con tres argumentos. Aunque la definición de `append` mostrada en este libro acepta solo dos argumentos, Scheme estándarmente proporciona un procedimiento `append` que toma un número arbitrario de argumentos.

43. Hemos usado el mismo símbolo `+` aquí para denotar tanto el procedimiento del lenguaje fuente como la operación de la máquina. En general, no habrá una correspondencia uno a uno entre las primitivas del lenguaje fuente y las primitivas de la máquina.

44. Hacer de las primitivas palabras reservadas es en general una mala idea, ya que un usuario no puede entonces reasignar estos nombres a diferentes procedimientos. Además, si agregamos palabras reservadas a un compilador que está en uso, los programas existentes que definen procedimientos con estos nombres dejarán de funcionar. Ver Ejercicio 5.44 para ideas sobre cómo evitar este problema.

45. Esto no es cierto si permitimos definiciones internas, a menos que las exploremos. Ver Ejercicio 5.43.

46. Esta es la modificación a la búsqueda de variables requerida si implementamos el método de escaneo para eliminar las definiciones internas (Ejercicio 5.43). Necesitaremos eliminar estas definiciones para que el direccionamiento léxico funcione.

47. Las direcciones léxicas no se pueden usar para acceder a variables en el entorno global, porque estos nombres pueden definirse y redefinirse interactivamente en cualquier momento. Con las definiciones internas escaneadas, como en el Ejercicio 5.43, las únicas definiciones que ve el compilador son las de nivel superior, que actúan sobre el entorno global. La compilación de una definición no hace que el nombre definido se ingrese en el entorno de tiempo de compilación.

48. Por supuesto, los procedimientos compilados, así como los interpretados,

son compuestos (no primitivos). Para ser compatibles con la terminología utilizada en el evaluador de control explícito, en esta sección usaremos "compuesto" para significar interpretado (en lugar de compilado).

49. Ahora que la máquina evaluadora comienza con un `branch`, siempre debemos inicializar el registro `flag` antes de iniciar la máquina evaluadora. Para iniciar la máquina en su bucle de lectura-evaluación-impresión ordinario, podríamos usar

(define (start-eceval) (set! the-global-environment (setup-environment)) (set-register-contents! eceval 'flag false) (start eceval))

50. Dado que un procedimiento compilado es un objeto que el sistema puede intentar imprimir, también modificamos la operación de impresión del sistema `user/print` (de Section 4.1.4) para que no intente imprimir los componentes de un procedimiento compilado:

(define (user-print object) (cond ((compound-procedure? object) (display (list 'compound-procedure (procedure-parameters object) (procedure-body object) '<procedure-env>))) ((compiled-procedure? object) (display '<compiled-procedure>)) (else (display object))))

51. Podemos hacerlo aún mejor extendiendo el compilador para permitir que el código compilado llame a procedimientos interpretados. Ver Ejercicio 5.47.

52. Independientemente de la estrategia de ejecución, incurrimos en una sobrecarga significativa si insistimos en que los errores encontrados en la ejecución de un programa de usuario sean detectados y señalados, en lugar de permitir que maten el sistema o produzcan respuestas incorrectas. Por ejemplo, una referencia a un arreglo fuera de los límites puede detectarse verificando la validez de la referencia antes de realizarla. Sin embargo, la sobrecarga de la verificación puede ser muchas veces el costo de la referencia al arreglo en sí, y un programador debe sopesar la velocidad contra la seguridad al determinar si tal verificación es deseable. Un buen compilador debería ser capaz de producir código con tales verificaciones, debería evitar verificaciones redundantes, y debería permitir a los programadores controlar el alcance y el tipo de verificación de errores en el código compilado. Los compiladores para lenguajes populares, como C y C++, apenas ponen operaciones de verificación de errores en el código en ejecución, para hacer que las cosas funcionen lo más rápido posible. Como resultado, recae en los programadores proporcionar explícitamente la verificación de errores. Desafortunadamente, la gente a menudo se olvida de hacerlo, incluso en aplicaciones críticas donde la velocidad no es una restricción. Sus programas

llevan vidas rápidas y peligrosas. Por ejemplo, el notorio "Gusano" que paralizó Internet en 1988 explotó la falla del sistema operativo UNIX(tm) para verificar si el búfer de entrada se ha desbordado en el demonio finger. (Ver Spafford 1989.)

53. Por supuesto, con cualquiera de las estrategias de interpretación o compilación también debemos implementar para la nueva máquina la asignación de almacenamiento, la entrada y la salida, y todas las diversas operaciones que tomamos como "primitivas" en nuestra discusión del evaluador y el compilador. Una estrategia para minimizar el trabajo aquí es escribir tantas de estas operaciones como sea posible en Lisp y luego compilarlas para la nueva máquina. En última instancia, todo se reduce a un pequeño núcleo (como la recolección de basura y el mecanismo para aplicar las primitivas reales de la máquina) que se codifica a mano para la nueva máquina.

54. Esta estrategia conduce a pruebas divertidas de la corrección del compilador, como verificar si la compilación de un programa en la nueva máquina, usando el compilador compilado, es idéntica a la compilación del programa en el sistema Lisp original. Rastrear el origen de las diferencias es divertido pero a menudo frustrante, porque los resultados son extremadamente sensibles a detalles minúsculos.

Referencias

Abelson, Harold, Andrew Berlin, Jacob Katzenelson, William McAllister, Guillermo Rozas, Gerald Jay Sussman y Jack Wisdom. 1992. The Supercomputer Toolkit: Un marco general para computación de propósito especial. *International Journal of High-Speed Electronics* 3(3): 337-361. –› **Allen**, John. 1978. *Anatomía de Lisp*. Nueva York: McGraw-Hill. **ansi** X3.226-1994. *Estándar Nacional Americano para Sistemas de Información—Lenguaje de Programación—Common Lisp*. **Appel**, Andrew W. 1987. La recolección de basura puede ser más rápida que la asignación en pila. *Information Processing Letters* 25(4): 275-279. –› **Backus**, John. 1978. ¿Puede la programación ser liberada del estilo von Neumann? *Communications of the acm* 21(8): 613-641. –› **Baker**, Henry G., Jr. 1978. Procesamiento de listas en tiempo real en una computadora en serie. *Communications of the acm* 21(4): 280-293. –› **Batali**, John, Neil Mayle, Howard Shrobe, Gerald Jay Sussman y Daniel Weise. 1982. La arquitectura Scheme-81—Sistema y chip. En *Actas de la Conferencia del mit sobre Investigación Avanzada en vlsi*, editado por Paul Penfield, Jr. Dedham, MA: Artech House. **Borning**, Alan. 1977. ThingLab—Un sistema orientado a objetos para construir simulaciones usando restricciones. En *Actas de la 5ª Conferencia Internacional Conjunta sobre Inteligencia Artificial*. –› **Borodin**, Alan, e Ian Munro. 1975. *La Complejidad Computacional de Problemas Algebraicos y Numéricos*. Nueva York: American Elsevier. **Chaitin**, Gregory J. 1975. Randomness and mathematical proof. *Scientific American* 232(5): 47-52. –› **Church**, Alonzo. 1941. *The Calculi of Lambda-Conversion*. Princeton, N.J.: Princeton University Press. **Clark**, Keith L. 1978. Negation as failure. In *Logic and Data Bases*. New York: Plenum Press, pp. 293-322. –› **Clinger**, William. 1982. Nondeterministic call by need is neither lazy nor by name. In *Proceedings of the acm Symposium on Lisp and Functional Programming*, pp. 226-234. **Clinger**, William, and Jonathan Rees. 1991. Macros that work. In *Proceedings of the 1991 acm Conference on Principles of Programming Languages*, pp. 155-162. –› **Colmerauer** A., H. Kanoui, R. Pasero, and P. Roussel. 1973. Un syst@ ‘eme de communication homme-machine en fran@ ,cais. Technical report, Groupe Intelligence Artificielle, Université d’Aix Marseille, Luminy. –› **Cormen**, Thomas, Charles Leiserson, and Ronald Rivest. 1990. *Introduction to Algorithms*. Cambridge, MA: MIT Press. **Darlington**, John, Peter Henderson, and David Turner. 1982. *Functional Programming and Its Applications*. New York: Cambridge University

Press. **Dijkstra**, Edsger W. 1968a. The structure of the "THE" multiprogramming system. *Communications of the acm* 11(5): 341-346. –› **Dijkstra**, Edsger W. 1968b. Cooperating sequential processes. In *Programming Languages*, edited by F. Genuys. New York: Academic Press, pp. 43-112. –› **Dinesman**, Howard P. 1968. *Superior Mathematical Puzzles*. New York: Simon and Schuster. **deKleer**, Johan, Jon Doyle, Guy Steele, and Gerald J. Sussman. 1977. AMORD: Explicit control of reasoning. In *Proceedings of the acm Symposium on Artificial Intelligence and Programming Languages*, pp. 116-125. –› **Doyle**, Jon. 1979. A truth maintenance system. *Artificial Intelligence* 12: 231-272. –› **Feigenbaum**, Edward, and Howard Shrobe. 1993. The Japanese National Fifth Generation Project: Introduction, survey, and evaluation. In *Future Generation Computer Systems*, vol. 9, pp. 105-117. –› **Feeley**, Marc. 1986. Deux approches à l'implantation du language Scheme. Masters thesis, Université de Montréal. –› **Feeley**, Marc and Guy Lapalme. 1987. Using closures for code generation. *Journal of Computer Languages* 12(1): 47-66. –› **Feller**, William. 1957. *An Introduction to Probability Theory and Its Applications*, volume 1. New York: John Wiley & Sons. **Fenichel**, R., and J. Yochelson. 1969. A Lisp garbage collector for virtual memory computer systems. *Communications of the acm* 12(11): 611-612. –› **Floyd**, Robert. 1967. Nondeterministic algorithms. *jacm*, 14(4): 636-644. –› **Forbus**, Kenneth D., and Johan deKleer. 1993. *Building Problem Solvers*. Cambridge, MA: MIT Press. **Friedman**, Daniel P., and David S. Wise. 1976. CONS should not evaluate its arguments. In *Automata, Languages, and Programming: Third International Colloquium*, edited by S. Michaelson and R. Milner, pp. 257-284. –› **Friedman**, Daniel P., Mitchell Wand, and Christopher T. Haynes. 1992. *Essentials of Programming Languages*. Cambridge, MA: MIT Press/ McGraw-Hill. **Gabriel**, Richard P. 1988. The Why of *Y*. *Lisp Pointers* 2(2): 15-25. –› **Goldberg**, Adele, and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Reading, MA: Addison-Wesley. –› **Gordon**, Michael, Robin Milner, and Christopher Wadsworth. 1979. *Edinburgh lcf*. Lecture Notes in Computer Science, volume 78. New York: Springer-Verlag. **Gray**, Jim, and Andreas Reuter. 1993. *Transaction Processing: Concepts and Models*. San Mateo, CA: Morgan-Kaufman. **Green**, Cordell. 1969. Application of theorem proving to problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 219-240. –› **Green**, Cordell, and Bertram Raphael. 1968. The use of theorem-proving techniques in question-answering systems. In *Proceedings of the acm National Conference*, pp. 169-181. –› **Griss**, Martin L. 1981. Portable Standard Lisp, a brief overview. Utah Symbolic Computation Group Operating Note 58, University of Utah. **Guttag**, John V. 1977. Abstract data types and the development of data structures. *Communications of the acm* 20(6): 396-404. –› **Hamming**, Richard W. 1980. *Coding and Information Theory*. Englewood Cliffs, N.J.: Prentice-Hall. **Hanson**, Christopher P. 1990. Efficient stack allocation for tail-recur-sive languages. In *Proceedings of acm*

Conference on Lisp and Functional Programming, pp. 106-118. –› **Hanson**, Christopher P. 1991. A syntactic closures macro facility. *Lisp Pointers*, 4(3). –› **Hardy**, Godfrey H. 1921. Srinivasa Ramanujan. *Proceedings of the London Mathematical Society* XIX(2). **Hardy**, Godfrey H., and E. M. Wright. 1960. *An Introduction to the Theory of Numbers*. 4th edition. New York: Oxford University Press. –› **Havender**, J. 1968. Avoiding deadlocks in multi-tasking systems. *ibm Systems Journal* 7(2): 74-84. **Hearn**, Anthony C. 1969. Standard Lisp. Technical report AIM-90, Artificial Intelligence Project, Stanford University. –› **Henderson**, Peter. 1980. *Functional Programming: Application and Implementation*. Englewood Cliffs, N.J.: Prentice-Hall. **Henderson**, Peter. 1982. Functional Geometry. In *Conference Record of the 1982 acm Symposium on Lisp and Functional Programming*, pp. 179-187. –›, 2002 version –› **Hewitt**, Carl E. 1969. PLANNER: A language for proving theorems in robots. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 295-301. –› **Hewitt**, Carl E. 1977. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence* 8(3): 323-364. –› **Hoare**, C. A. R. 1972. Proof of correctness of data representations. *Acta Informatica* 1(1). **Hodges**, Andrew. 1983. *Alan Turing: The Enigma*. New York: Simon and Schuster. **Hofstadter**, Douglas R. 1979. *Gödel, Escher, Bach: An Eternal Golden Braid*. New York: Basic Books. **Hughes**, R. J. M. 1990. Why functional programming matters. In *Research Topics in Functional Programming*, edited by David Turner. Reading, MA: Addison-Wesley, pp. 17-42. –› **ieee** Std 1178-1990. 1990. *ieee Standard for the Scheme Programming Language*. **Ingerman**, Peter, Edgar Irons, Kirk Sattley, and Wallace Feurzeig; assisted by M. Lind, Herbert Kanner, and Robert Floyd. 1960. THUNKS: A way of compiling procedure statements, with some comments on procedure declarations. Unpublished manuscript. (Also, private communication from Wallace Feurzeig.) **Kaldewaij**, Anne. 1990. *Programming: The Derivation of Algorithms*. New York: Prentice-Hall. **Knuth**, Donald E. 1973. *Fundamental Algorithms*. Volume 1 of *The Art of Computer Programming*. 2nd edition. Reading, MA: Addison-Wesley. **Knuth**, Donald E. 1981. *Seminumerical Algorithms*. Volume 2 of *The Art of Computer Programming*. 2nd edition. Reading, MA: Addison-Wesley. **Kohlbecker**, Eugene Edmund, Jr. 1986. Syntactic extensions in the programming language Lisp. Ph.D. thesis, Indiana University. –› **Konopasek**, Milos, and Sundaresan Jayaraman. 1984. *The TK! Solver Book: A Guide to Problem-Solving in Science, Engineering, Business, and Education*. Berkeley, CA: Osborne/McGraw-Hill. **Kowalski**, Robert. 1973. Predicate logic as a programming language. Technical report 70, Department of Computational Logic, School of Artificial Intelligence, University of Edinburgh. –› **Kowalski**, Robert. 1979. *Logic for Problem Solving*. New York: North-Holland. –› **Lamport**, Leslie. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the acm* 21(7): 558-565. –› **Lampson**, Butler, J. J. Horning, R. London, J. G. Mitchell, and G. K. Popek.

1981. Report on the programming language Euclid. Technical report, Computer Systems Research Group, University of Toronto. –› **Landin**, Peter. 1965. A correspondence between Algol 60 and Church's lambda notation: Part I. *Communications of the acm* 8(2): 89-101. **Lieberman**, Henry, and Carl E. Hewitt. 1983. A real-time garbage collector based on the lifetimes of objects. *Communications of the acm* 26(6): 419-429. –› **Liskov**, Barbara H., and Stephen N. Zilles. 1975. Specification techniques for data abstractions. *ieee Transactions on Software Engineering* 1(1): 7-19. –› **McAllester**, David Allen. 1978. A three-valued truth-maintenance system. Memo 473, MIT Artificial Intelligence Laboratory. –› **McAllester**, David Allen. 1980. An outlook on truth maintenance. Memo 551, MIT Artificial Intelligence Laboratory. –› **McCarthy**, John. 1960. Recursive functions of symbolic expressions and their computation by machine. *Communications of the acm* 3(4): 184-195. –› **McCarthy**, John. 1963. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, edited by P. Braffort and D. Hirschberg. North-Holland. –› **McCarthy**, John. 1978. The history of Lisp. In *Proceedings of the acm sigplan Conference on the History of Programming Languages*. –› **McCarthy**, John, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. 1965. *Lisp 1.5 Programmer's Manual*. 2nd edition. Cambridge, MA: MIT Press. –› **McDermott**, Drew, and Gerald Jay Sussman. 1972. Conniver reference manual. Memo 259, MIT Artificial Intelligence Laboratory. –› **Miller**, Gary L. 1976. Riemann's Hypothesis and tests for primality. *Journal of Computer and System Sciences* 13(3): 300-317. –› **Miller**, James S., and Guillermo J. Rozas. 1994. Garbage collection is fast, but a stack is faster. Memo 1462, MIT Artificial Intelligence Laboratory. –› **Moon**, David. 1978. MacLisp reference manual, Version 0. Technical report, MIT Laboratory for Computer Science. –› **Moon**, David, and Daniel Weinreb. 1981. Lisp machine manual. Technical report, MIT Artificial Intelligence Laboratory. –› **Morris**, J. H., Eric Schmidt, and Philip Wadler. 1980. Experience with an applicative string processing language. In *Proceedings of the 7th Annual acm sigact/sigplan Symposium on the Principles of Programming Languages*. **Phillips**, Hubert. 1934. *The Sphinx Problem Book*. London: Faber and Faber. **Pitman**, Kent. 1983. The revised MacLisp Manual (Saturday evening edition). Technical report 295, MIT Laboratory for Computer Science. –› **Rabin**, Michael O. 1980. Probabilistic algorithm for testing primality. *Journal of Number Theory* 12: 128-138. **Raymond**, Eric. 1993. *The New Hacker's Dictionary*. 2nd edition. Cambridge, MA: MIT Press. –› **Raynal**, Michel. 1986. *Algorithms for Mutual Exclusion*. Cambridge, MA: MIT Press. **Rees**, Jonathan A., and Norman I. Adams IV. 1982. T: A dialect of Lisp or, lambda: The ultimate software tool. In *Conference Record of the 1982 acm Symposium on Lisp and Functional Programming*, pp. 114-122. –› **Rees**, Jonathan, and William Clinger (eds). 1991. The $\rm revisedˆ4$ report on the algorithmic language Scheme. *Lisp Pointers*, 4(3). –› **Rivest**, Ronald, Adi

Shamir, and Leonard Adleman. 1977. A method for obtaining digital signatures and public-key cryptosystems. Technical memo LCS/TM82, MIT Laboratory for Computer Science. –› **Robinson**, J. A. 1965. A machine-oriented logic based on the resolution principle. *Journal of the acm* 12(1): 23. **Robinson**, J. A. 1983. Logic programming—Past, present, and future. *New Generation Computing* 1: 107-124. **Spafford**, Eugene H. 1989. The Internet Worm: Crisis and aftermath. *Communications of the acm* 32(6): 678-688. –› **Steele**, Guy Lewis, Jr. 1977. Debunking the "expensive procedure call" myth. In *Proceedings of the National Conference of the acm*, pp. 153-62. –› **Steele**, Guy Lewis, Jr. 1982. An overview of Common Lisp. In *Proceedings of the acm Symposium on Lisp and Functional Programming*, pp. 98-107. **Steele**, Guy Lewis, Jr. 1990. *Common Lisp: The Language*. 2nd edition. Digital Press. –› **Steele**, Guy Lewis, Jr., and Gerald Jay Sussman. 1975. Scheme: An interpreter for the extended lambda calculus. Memo 349, MIT Artificial Intelligence Laboratory. –› **Steele**, Guy Lewis, Jr., Donald R. Woods, Raphael A. Finkel, Mark R. Crispin, Richard M. Stallman, and Geoffrey S. Goodfellow. 1983. *The Hacker's Dictionary*. New York: Harper & Row. –› **Stoy**, Joseph E. 1977. *Denotational Semantics*. Cambridge, MA: MIT Press. **Sussman**, Gerald Jay, and Richard M. Stallman. 1975. Heuristic techniques in computer-aided circuit analysis. *ieee Transactions on Circuits and Systems* CAS-22(11): 857-865. –› **Sussman**, Gerald Jay, and Guy Lewis Steele Jr. 1980. Constraints—A language for expressing almost-hierachical descriptions. *AI Journal* 14: 1-39. –› **Sussman**, Gerald Jay, and Jack Wisdom. 1992. Chaotic evolution of the solar system. *Science* 257: 256-262. –› **Sussman**, Gerald Jay, Terry Winograd, and Eugene Charniak. 1971. Microplanner reference manual. Memo 203A, MIT Artificial Intelligence Laboratory. –› **Sutherland**, Ivan E. 1963. SKETCHPAD: A man-machine graphical communication system. Technical report 296, MIT Lincoln Laboratory. –› **Teitelman**, Warren. 1974. Interlisp reference manual. Technical report, Xerox Palo Alto Research Center. –› **Thatcher**, James W., Eric G. Wagner, and Jesse B. Wright. 1978. Data type specification: Parameterization and the power of specification techniques. In *Conference Record of the Tenth Annual acm Symposium on Theory of Computing*, pp. 119-132. **Turner**, David. 1981. The future of applicative languages. In *Proceedings of the 3rd European Conference on Informatics*, Lecture Notes in Computer Science, volume 123. New York: Springer-Verlag, pp. 334-348. **Wand**, Mitchell. 1980. Continuation-based program transformation strategies. *Journal of the acm* 27(1): 164-180. –› **Waters**, Richard C. 1979. A method for analyzing loop programs. *ieee Transactions on Software Engineering* 5(3): 237-247. **Winograd**, Terry. 1971. Procedures as a representation for data in a computer program for understanding natural language. Technical report AI TR-17, MIT Artificial Intelligence Laboratory. –› **Winston**, Patrick. 1992. *Artificial Intelligence*. 3rd edition. Reading, MA: Addison-Wesley. **Zabih**, Ramin, David McAllester, and David Chapman. 1987. Non-deterministic Lisp

with dependency-directed backtracking. *aaai-87*, pp. 59-64. → **Zippel**, Richard. 1979. Probabilistic algorithms for sparse polynomials. Ph.D. dissertation, Department of Electrical Engineering and Computer Science, MIT. **Zippel**, Richard. 1993. *Effective Polynomial Computation*. Boston, MA: Kluwer Academic Publishers.

Lista de Ejercicios

Lista de Figuras

Índice

> Cualquier inexactitud en este índice puede explicarse por el hecho* de que ha sido preparado con la ayuda de un ordenador. —Donald E. Knuth, *Algoritmos Fundamentales** (Volumen 1 de *El Arte de Programar Computadoras*)

@end ifinfo

Colofón

EN LA PORTADA se encuentra el mecanismo de rueda de libros de Agostino Ramelli de 1588. Podría verse como una ayuda temprana para la navegación en hipertexto. Esta imagen del grabado está alojada por J. E. Johnson de New Gottland. Las tipografías son Linux Libertine para el texto del cuerpo y Linux Biolinum para los encabezados, ambas de Philipp H. Poll. La tipografía de máquina de escribir es Inconsolata creada por Raph Levien y complementada por Dimosthenis Kaponis y Takashi Tanigawa en la forma de Inconsolata LGC. La tipografía de la portada es Alegreya, diseñada por Juan Pablo del Peral. El diseño gráfico y la tipografía están a cargo de Andres Raba. El código fuente de Texinfo se convierte a LaTeX mediante un script de Perl y se compila a PDF con XeLaTeX. Los diagramas se dibujan con Inkscape.